The dynamic of modern software development project management and the software crisis of quality

An integrated system dynamics approach towards software quality improvement

Armindokht Nasirikaljahi

Submitted in Partial Fulfillment of the Requirement for the Degree of

Master of Philosophy in System Dynamics

System Dynamics Group

Department of Geography

UNIVERSITY OF BERGEN,

BERGEN

NORWAY

July, 2012

Acknowledgements

I would like to take this opportunity to give my sincere thanks to my professors, colleagues and friends that have been supporting me through this research-effort.

First of all, I would like to thank my supervisor Professor Pål I. Davidsen. Under your supervision and guidance I was able to develop and complete this thesis. During the project, I was given valuable feedback and encouragement. I would also like to thank you for introducing me to Abdel-Hamid's model and the opportunity to develop it further. The model allowed me to pursue my interest in the topic of software quality with a system dynamics framework. This opportunity was valuable for me with my background in IT-engineering.

I would also like to thank Professor Erling Moxnes and Associate Professor, Dr. I. David Wheat. The courses I attended during the system dynamics master program improved my understanding and skills of systems management. Their effort as lecturers are truly remarkable and valuable for any student who wish to explore the opportunities of dynamic systems thinking.

My gratitude is given to my good friends and colleagues in the System Dynamics Group. Thank you Jahanzeb Muhammad, and Richard McDowall. Whenever I had a technical problem or a methodological question I could really count on you to provide me with a good and sound answer. This support was very valuable for me.

Last but certainly not least I would like to give my gratitude to my dear and close friend Steinar Gjerde. Through our discussions and debates on the topic of software quality, I was inspired to seek alternative solutions to well-known problems. Some of these ideas are reflected in my paper as well. Most importantly though, your constant encouragement, positive energy, and friendship supported me through the long days of analysis and research. For that I thank you from the bottom of my heart.

Armindokht Nasirikaljahi

Bergen, Norway

July, 2012

Table of contents

List of figures

Abstract

The software industry is plagued by cost-overruns, delays, poor customer satisfaction and quality issues that are costing clients and customers world-wide billions of dollars each year. The phenomenon is coined "The Software Crisis", and poses a huge challenge for software project management. This thesis addresses one of the core issues of the software crisis, namely software quality. The challenges of software quality are central for understanding the other symptoms of the software crisis. The dynamics of software quality will be examined through a system dynamics approach in order to reveal the behavioral patterns behind the managerial challenges. This paper utilizes the pioneering model of Tarek Abdel-Hamid that provides the core systems and behaviors of software project management. By further developing this model, I provide three core enhancements that will provide software management with additional information into the challenges of software quality and concrete policy solutions to achieve greater software quality. The enhancements include a dynamic and milestone based testing system, a client review scheme and a "Capture Re-capture" pre-test system. These systems will provide overall greater software quality within reasonable cost-levels. They also provide an additional aid for managers and clients to construct more realistic plans concerning schedules and manpower-allocation for the software development cycle.

Key words: software project management, software crisis, cost overruns, delays, software quality, managerial challenges, system dynamics, software testing, client-review, capture re-capture.

Thesis core review

This thesis is a comprehensive study into the software crisis, and the question of software quality. It deploys as a foundation the pioneering system dynamic model of Tarek Abdel-Hamid published in his doctoral thesis in 1991. My aim to improve his model and contribute to increased software quality provides a large width and scope to my thesis. As a result this thesis consists of several chapters containing important information, foundations for development and dynamic analyses. In light of this depth, I provide this core review. It contains information on the chapters found in my thesis, and the important information they provide. Hence, this review functions as a quick-reference for the reader, where the core description of each chapter is provided.

Chapter 1: Introduction

The first chapter of my thesis is the introductory chapter, which consists of two main parts. The first one describes the characteristics of the software crisis that inspired Abdel Hamid's original thesis. The software crisis held huge financial losses for the client-side during the 70s and 80s. Reports from the US congress in 1981 described the situation as a constant system of low software quality, high costs, budget overruns and long delays.

The second part of chapter 1 describes the software crisis in modern times. Through articles in IEE published in the 2000s and reports from the EU during the last decade, the software crisis is established as a continuing phenomenon. The same characteristics described in the 80s are found in the recent description of the software crisis. Software often hold low quality, are riddled with errors, are significantly delayed and projects often overrun the budget.

Chapter 2: Literature review

The second chapter contains the literature review for my thesis. In order to build a solid theoretical foundation, I have reviewed the core literature on the field. The chapter consists of four parts. The first part establishes the traditional publications on software project management. This literature is the foundation for Abdel-Hamid's original model. The second part of the literature review is a presentation of system dynamics papers that incorporate the original model in their foundation. These papers all establish the strong theoretical foundation of the original model, and that the concept described within are utilized in further research. The importance of the model in the scientific tradition of system dynamics is established by these papers. They also provide the foundation of project management and software project management in the system dynamics tradition. The third part of the literature review focus on the quality aspect of software production. In order to establish the importance of quality, and how to obtain it, I present the traditional school of quality thinking established in the concept of total quality management. The fourth part of the literature review establishes the theoretical importance of the client in production, and the concept of software quality in a selection of modern system dynamics publications.

Chapter 3: Research Method

The third chapter provides the methodological aspects of the system dynamics approach. In this chapter I explain the core concepts of the system dynamics modeling, and the tools deployed. I explain the concept of causal loop diagrams, how they depict the causal relationships between variables and the system of feedback loops. The core visual aids for such a diagram are then explained. I continue with an explanation of the stock and flow diagram, and the logic behind them. I present the visual symbols for such diagrams so the concept is clear when I present both stock and flow diagrams and causal loop diagrams later in my analysis. The chapter ends with a description of both delays and non-linearity, and how these aspects are important concepts for the system dynamics method.

Chapter 4: The structure of software development project management

The fourth chapter is a particularly important chapter in my thesis and contains one of the most important contributions. In this chapter I provide the dynamic relationship between the variables inside the subsector of the original model. The original model is unfortunately not transparent on this issue. It is opaque in this department, as major feedback loops are not exhibited. Graphically the model has been displayed component by component in the form of stock and flow diagrams. These diagrams do not convey the overall structure of the model, nor do they provide the cross-subsystem relationships between variables. Therefore one of the purposes of my thesis is to display this overall structure as a means to make the original model interpretable. I then utilize this exposition as a base to introduce structure modifications in the original model. The chapter contains causal loop diagrams for all subsystems, and the dynamic relationship between them, both inter- and intra-subsystem.

Chapter 5: Research Questions and Reference Mode

The fifth chapter contains my research question and the reference mode. For my thesis there are two main questions that I seek to answer through my analysis. The first research question is formulated as: ”*Are there any unrealistic assumptions in Tarek Abdel-Hamid's "Software Development Project management" model? If so, how we can modify those assumptions to make the model more realistic?*”.

The second research question is formulated as: “*In terms of policy conclusion (for example: distribution used to resources), are there additional policies for instance, supplementary allocation of resources to quality assurance, rework and testing efforts that would improve the model behaviour?”*

The second part of chapter 5 presents the reference mode of the software crisis issue, and how it is graphically and numerically depicted through key elements of the model. The aim for a system dynamics model is the ability to reproduce the characteristics of a problem before aiming to solve or improve the situation. This is presented in detail in this chapter.

Chapter 6: Dynamic hypothesis and model structure

The sixth chapter is another chapter of great importance in my research. In this chapter I present my model enhancements. These are constructed on the basis of the causal loop diagrams from chapter 4 and the empirical descriptive data of modern software projects. This chapter introduces important enhancements in regards to both realism and improvement of software quality. The model enhancement boundary and the time horizon are established. The enhancements to the rework effort in regards to error generation and manpower allocation are introduced, as well as my enhanced testing regime and the capture re-capture effort. The latter enhancements are major contributions in my thesis and form the foundation of policy decisions analysed in chapter 8. In the original model, the testing regime was constructed according to the practises at the end of the 80s. In modern software project these tools have evolved and my enhanced testing effort along with the concept of capture re-capture provide needed realism for a software project model at the beginning of the 2010s. All the enhancements are introduced in depth with causal loop diagrams, and stock and flow diagrams for each policy enhancement.

Chapter 7: Model validation and analysis

The seventh chapter of my thesis contains the model validation. This is an important process, where the validity of my model is thoroughly scrutinized. Through several techniques, including both indirect theoretical/empirical methods and direct model tests, the validity of my model is strengthened. In addition, the robustness of my results increases as the model's strength and validity is established. The tests contain techniques such as the structure test, the parameter test, the extreme condition test and the integration error test.

One of my main efforts in chapter 7 is a comprehensive loop knock-out analysis. Here I cut the effect of central loops in the software model. This enables me to verify the validity of the causal loops I constructed in chapter 4. Additionally the test allows me to investigate the strength of the loops across time during a simulation. This test would have been impossible to carry out without the construction of causal loops in chapter 4.

Chapter 8: Policy Analysis

The eighth chapter contains the policy analysis of my thesis, and is another very important chapter. Here I analyse the possible policy-strategies management may deploy in order to provide software production quality, and quality for the end product. My aim is to contribute to the enhancement of overall software quality, and this is achievable through the policy-decisions I provide in this chapter. There are three main areas of software development that determines the software quality and success of the project. The allocation of manpower to these efforts is the determining factor and engine behind my policy suggestions. The software crisis is a managerial problem, and my policy suggestions will provide an aid for software project managers when allocation of manpower is concerned.

Chapter 9: Conclusion

The final chapter of my thesis contains my conclusion. Here I return to my research question, and investigates the answers my analysis has provided. The model has revealed the need for enhancements and development of the original model. The implications of these enhancements do provide additional allocation policies that change the rate of resource allocation to quality assurance, rework and testing.

Chapter 1: Introduction

1.0 Introduction

During the last decades there has been a significant evolution within the software industry. From being an industry largely related to military development, or resource-demanding governmental systems such as aviation, it now provides products to a huge segment of different markets and areas. The IT revolution of the 90's and early 20s broadened the scope considerably, and as Robert Charette puts it in an article for IEEE: "Software is everywhere" (Charette, 2010).

Today software is an integrated part of our daily lives. They run our computers, cell phones, cars and different household appliances. In society software is responsible for controlling our financial banking system, our hospitals and traffic grids. Large governmental programs depend on IT and software products in order to fulfill their duties. Thus governments are an important source of software demands. In 2003 the UK government reported more than 100 major IT projects underway. The total value of UK contracts reached \$20.3 billion. In 2004 the US government cataloged 1200 civilian IT projects, costing more than 60 billion dollars. It is clear from the statistics that substantial government money is invested in IT and software projects (Charette, 2010).

The EU is also an important actor for IT investments. The union is continuously expanding on the European continent, and the demand for software to run large-scale pan-European programs is increasing. The ICT sector has grown considerably the last 10 years, and the EU cites software and IT one of the main forces for integration. The common market and cross-European standardization had been impossible without heavy ICT investments. (ESSU: 2000)

The growth of the software however, has not been a one-sided success. With more investments and larger demands several key problems have emerged and manifested themselves. As early as the late 70s Pressman observed that the industry was indeed making a transition, but the growth came with circumstances that indicated a "crisis". The records show that the software industry has been marked by cost-overruns, late deliveries, poor reliability and user's dissatisfaction. These hallmarks are all part of a larger phenomenon named the "software crisis" (Abdel-Hamid, 1991: p.3).

1.1. Early signs of the software crisis

As early as 1979 the term “Software crisis” was coined in the public debate. A congressional report issued by the controller general cited the dimension of a software crisis in the federal government. The headline of the report summarized the issue: “Controlling of Computer Software Development. Serious Problems Require Management Attention to Avoid Wasting Additional Millions.” The report concluded that the US government got less than 2% of the total value of contracts for its investment (Abdel-Hamid, 1991: p.3).

One decade later the problem was identified again. In an issue of Defense News for example, the situation concerning the Peace Shield project was scrutinized. The project was at the time running four years behind schedule and over-running budget by an estimation of $300 million.

The reports on the software crisis during the late 70s and 80s also revealed a trend of lacking quality in the software products. When the final product was introduced, it often contained significant errors that would lead to additional maintenance or development. This would push the costs and delays even further, or lead to scrapping of software right after delivery.

The congressional report of 1979 provides examples of low quality in delivered products to the military. There are several other papers published, by for example Thayer (1986), Frank (1983) and Schlender (1989), that mention the lack of quality in the software end-product. However, in the early stages of the software crisis the focus was mainly on software cost overruns and delays. When managers did turn to the question of consumer satisfaction, the focus moved towards software quality.

Paulk observes that at the same time as organizations where focusing on customer satisfaction, there was a growing perception that software quality was the weak link in developing high quality products (Paulk, 1993). However, the symptoms of the software crisis that attracted the most of attention moved focus away from quality as the cause of the disease. Therefore the literature on the early signs of the software crisis mention quality as a side, but focus remains on cost and management.

The reports on the software crisis fuelled a debate within the industry itself and in the academic field. Scholars and technicians started to focus on the industry's own development-practices in order to the mechanisms causing the problem and solutions to them. No other industry had at the time been able to survive with such deplorable return rates over a long time period. The industry itself was quite young at the time, and therefore the tools for analysis and management was lacking. Therefore several pioneering studies were published in this time period, addressing the production system of the industry itself.

Notable contributions were made in this effort, for example Boehm's insightful analysis from TGW leading to the COCOMO analysis tool (Abdel-Hamid, 1991). In the beginning of the 90's Tarek Abdel-Hamid published his doctoral thesis on the dynamics inside a software production system. This thesis contains the first attempt to build a complete System Dynamics model in order to identify how the complexity of a production cycle fuels the software development crises. His thesis became a very influential publication, and served as a pioneering effort into the field of software development. The effort culminated in a book called *Software Project Dynamics: An Integrated Approach.* Later his model has been used as a base for further studies and development of core concepts within software development.

From his work two distinct views on the software crisis emerges: The first consideration of the situation is managerial. The lack of routines and discipline within the industry itself is causing poor management decisions. The root of the crisis is therefore a general lack of management efforts in the different stages of development that causes poor planning, estimation of costs and staffing strategies. This in turn causes significant problems when circumstances in development demand more attention than perceived (Abdel-Hamid, 1991: pp. 3-5).

The second view is concerning the technical side of the problem. This position argued that the reason for software delays and cost overruns stemmed from technical hurdles. In a young industry the technical tools of the trade was being developed during production, hence creating delays and errors from trial and error (Abdel-Hamid, 1991: p.6).

The second considerations attracted a lot more attention, and significant inroads were made to tackle the technical hurdles. Despite their effort, and more readily available tools the problem from the software crisis continued to plague the industry.

1.2 The software crisis in modern times

Since the first reports of the software crisis emerged, three decades have passed. The software industry has gone through even more evolutionary steps, and the IT revolution of the late 90s and early 20s have changed the face of the industry completely. Today software is big business, and can be found anywhere. The industry has been able to transform its scope and width considerably. Unfortunately, the question of a software crisis is still an undeniable factor.

It becomes readily apparent that the software crisis looms still at large just as it did at the end of the 70's and 80's. Robert Charette's article in IEEE describes the situation, and the hallmarks for the industry are very similar to those reported in the congressional report. The industry is plagued by overruns, delays and terminations and the total value due to abandonment alone is staggering. The US spent in 2004 $60 billion in software contracts, and with a modest failure-rate reported by Charette at 5%, which would bring the loss to an estimation of $3 billion. However, the author speculates that the true fail-rate could be as high as 15-20% of all software contracts. These are contracts terminated or abandoned shortly before or after delivery. Charette believes that over the last five years bad software that had to be abandoned cost the US economy between $25 and $75 billion (Charette, 2010).

In December of 2007 Dexter Whitfield at the European Services Strategy Unit (ESSU) published a report on outsourced public sector IT projects. This report supports the findings made by Charette in the US, and its title even echoes that of the Congressional report of 1979. The ESSU report is named: "Cost overruns, Delays and Terminations: 105 outsourced public sector ICT projects" (ESSU, 2007).

The ESSU report shares many of the findings reported over the last decades. The ICT sector is hallmarked by delays, cost overruns and implementation problems. The total value of the contracts was £29.5 billion, while the total cost overruns totaled an additional £9 billion. 57% of all contracts experienced cost overruns at an average of 30.5%. In 33% of all the 105 programs one experienced major delays, and 30% were terminated (ESSU, 2007). The ESSU reports identify several areas that cause the costs to sore due to underestimation. These are linked to additional staffing due to technical issues or consolatory staff as the project passes through the different production stages. The report also concludes that the problems are caused by managerial shortcomings.

Just as in the earlier reports and publications on the software crisis, the concept of software quality emerges in the modern debate as well. Both Charette and the ESSU report provide ample examples of the lack of quality in the software end-product and how this creates software failures. Charette provides a “wall of shame” for the software industry. On this wall we find several large software failures generating huge losses. Among these examples we find the Hudson Bay Co. The Hudson Bay Corporation experienced serious problems with their inventory system named “Big Ticket”. The IT project was supposed to change almost every aspect of the IT infrastructure at the firm. The problem contributed to a loss of $33.3 million, and was largely attributed to poor software quality. The system was not able to operate the complex system it was set to manage (Charette, 2010).

The second example from the wall of shame concerns the UK Inland revenue. In 2004-05 the Inland Revenue office experienced software errors. The contractor, EDS, was set to construct an IT tax-credit system for poor families. Two reports from the Parliamentary Ombudsman and Citizens Advice reveal, describe the delivered software as abysmal. Nearly two million people were overpaid a total of £32 billion. On average, families were overpaid by £1000. This money was later reclaimed by the Inland Revenue, causing more financial problems for poor families as a result. Reports indicate that some families had to rely on food-parcels to get by (Charette, 2010). The UK government considered suing the EDS for the failed software, but later revoked the contract and gave it to CAP Gemini.

1.3 The software crisis of quality

Besides from being a crisis that inflicts huge costs on both manufacturer and clients, the second pattern that emerges from the observations on the software crisis is the lack of quality in the end-products. Jones and Bonsignour observe that while software is among the most widely used products in human history; it also has one of the highest failure rates of any product in human history. The reason for software failures is primarily due to poor quality (Jones & Bonsignour, 2012). Due to the fact that society itself is now heavily reliant on software to operate, the consequences of software failures can have dire consequences for their users beyond the cost-problem. If your car’s breaking system fails, or if a hospital makes a medical mistake there is a high probability that poor software quality was part of the problem.

In Norway we saw an example were the governmental portal for electronic deliverance of public forms and documents failed unexpectedly. For several hours, the full personal information regarding taxes and other records of a single individual was presented to any person logging in to the portal. The software project for the Altinn portal cost the Norwegian government approximately $165 million. After a crisis-meeting it became apparent that an additional $165 million would have to be spent in order to fix the problem permanently (NVE, 2012).

This example shows how software failure produces two kind of serious problems. One is obviously the cost of fixing the problem, the second is the fact that for a while classified or sensitive documents were made public. The question of quality in software goes beyond that of just costs.

Hjertø identifies several factors that high quality provides for the software industry. Higher quality enables the reduction the client's dissatisfaction. Higher quality awareness in the production-process itself allows us to reduce the error-rate, the rework effort and the scrap-rate. With an increased focus on quality during the production process, the costs for inspections and error-control will also decrease. Reduced errors and scrap-rates reduce the costs for post-production penalties and liability-claims due to failed products.

Aside from the cost-aspects Hjertø argues that a quality-oriented production-cycle will provide increased production-capacity and delivery accuracy. This is a factor of increased awareness and information through the cycle. Charette's criticism of the management-side of the industry addresses this issue. Managers are not able to create realistic estimates of costs and delivery. The reason for this following Hjertø's argument is a lack of information during the production process. She offers a paradox to the software industry: "There is never time to do things right the first time, but always time to redo the task later. We never have the money to prevent failures from appearing, however we always have the money to repair and fix errors over and over" (Hjertø, 2003).

This leaves a rather interesting conundrum to the software crisis. It is apparent that poor software quality and failed software projects are creating huge costs and losses to both industry and user. This problem has now been apparent for almost 40 years. However, when one look into the core of the problem, we see that arguments around quality often focus on the increased costs of providing better quality.

There is a second relationship within the software industry that is important when quality is being discussed. The relationship between the developer and the client is a highly debated issue in software development. The client is the end user of the product, and software programs are tailored to specific requirements. It is very important to understand the needs of clients or customers when designing and manufacturing software.

In recent years, we have seen practices develop that includes a wider use of end-user reviewing and beta-testing to ensure client satisfaction. In the earlier years of the industry, the heavy technical nature of computer development disqualified the client from participating unless trained in computer science. With newer programs, easier interfaces and more intuitive programming, clients and customers are able to test and comment on the finished product.

In the literature however, we see that there is a divided view on allowing clients into the production process. Some argue that clients are a good source for information, and will enable better quality in the end (Jones & Bonsignour, 2012). Software developers should act as the mediator between product and client, ensuring that the end-user determines the important functional factors (Hjertø, 2007: p.88). The other side of the argument deems the client unfit for understanding the finer points of software development. The client's ideas provide information, but cannot be the determinant factor for the software's overall quality. DeMarco and Lister al argues "*Allowing the standard of quality to be set by the buyer, rather than the builder, is what we call the flight from excellence*" They continue "*Quality, far beyond that required by the end user, is means to higher productivity"* (DeMarco & Lister, 1999).

In the original model of Abdel-Hamid, the client was excluded from the process itself. Based on the available literature today, this seems like an omission that needs to be addressed. There is a clear relationship between quality, costs and client. Understanding the potential for the client in this process provides a broader base for understanding the foundation of software quality. This captured my interest regarding the concept of the software crisis, as it provides a clear need for further research. The first main question that influences my approach is "How can we improve the overall quality of the software product, and how will this influence the mechanisms of the software crisis?"

Just as Abdel-Hamid was inspired to construct his system dynamics model on the software crisis, I am inspired to further develop the model in order to enhance our understanding of software quality and how to better achieve it. Since the original model provided the main sectors for software production and the different efforts within, I will use this model as my base. In the base model, quality control and improvement of the produced software hinges on three main efforts. These are Quality Assurance, Rework and Testing. My research will therefore look into these subsectors in order to seek policy-potentials regarding overall software quality.

This thesis organized as follows: In the second chapter I provide a literature review in order to establish the relationship between software production, quality and the system dynamics method. The third chapter the technical considerations and system dynamics methodology will be presented. A brief introduction of Abdel-Hamid's model, causal loops diagrams for each subsystem and subsector and dynamic of software development project management as it will be the base for my research will be presented in chapter 4. The fifth chapter my research questions, and the reference modes that are the important parameters concerning quality assurance, rework and testing efforts will be presented. Afterward my hypothesis and model structure will be presented in chapter 6 while providing a model validation and behavior analysis through chapter 7. In Chapter 8 I will provide policy analyses. In chapter 9 I will conduct testing on key behavior-problems from the base model on the enhanced model. These behaviors concern Brook's law, the 90% syndrome and economy of quality assurance. The 10th chapter will present my findings and a conclusion.

Chapter 2: Literature review

2.0 Introduction

In this chapter I will provide a review of some of the central literature and publications focusing on the topic for my research. The goal of this chapter is to provide a theoretical foundation for the essential aspects of my paper considering System Dynamics modeling as a methodology and the concept, dynamic, significance and crises of quality within software development.

The objective for this chapter is fourfold. Since I will be using Abdel-Hamid's model as a base for my research and analyses, relevant publications that forms the basis for the model will be presented in order to provide the logic behind its development. Secondly new publication will be presented in order to position the core substructures in the current project management field and system dynamic. The third objective is to provide a basis for the method I am applying, namely the concept of system dynamics linked to both managing software development and quality of software product. The fourth goal is to establish the concept, significance and crisis of quality within production and software production. The fifth goal is to present some publications and previous studies related to system dynamics and Quality.

2.1 Building blocks for Abdel-Hamid's model of software project management

In order to establish the essential building-blocks of the base-model, one has to return to the classic papers on both system dynamics modeling and project management. Edwards B. Robert's doctoral dissertation completed in 1962 was the first effort to apply System Dynamics as a method to project management within a research and development environment (Roberts, 1962).

Since the doctoral dissertation, Roberts have been playing an active role as a guide in System Dynamics advancing the field through studies of R&D. Since the software industry was a relative young industry, it was important to establish its relationship with other established fields. Through the fields of for example R&D, one could find both similarities and concepts to further develop the understanding of software management.

Gehring and Pooch (1977) noted that the stages of research and development are similar in many ways to the stages of software analysis and design. Wolverton's paper published in 1974 supports this notion, were he notes that the general principles of pricing large R&D efforts of any kind apply to large software development projects as well (Wolverton, 1974).

Other notable work that combines the features of R&D and Software development is Putnam's SLIM model (Putnam, 1964). The SLIM model for software cost estimation is based on Norden's R&D analysis (Norden, 1963). Norden discovered that R&D projects have a well-defined manpower pattern of the Rayleigh form. Putnam adapted Norden's findings and discovered that in the software environment manpower application follows the same Rayleigh Pattern.

Richard Thayer's Ph.D. dissertation from 1979 called "Modeling a Software Engineering Project Management System" is an important base for Abdel-Hamid's model. Thayer's work is the first attempt to completely model a software engineering project, and it generated a lot of attention when it was published. Thayer's goal was twofold, as he sought to develop and verify a generalized descriptive management model and to verify and identify major issues of software engineering management systems. Thayer's model breaks management of a software project into five functions: Planning, organizing, staffing, directing and controlling (Thayer, 1979).

Riehl's doctoral thesis from 1977 developed a "framework to assist in the management of computer-based information systems development in large organizations. (Riehl, 1977) Riehl's model termed the "Composite-Working model" consisted of 25 principles and 50 issues

McFarlan's work is different from Riehl and Thayer's, as he chose to focus on the differences of issues among software development projects rather than what they have in common. For McFarlan it was important to distinguish between different kinds of software projects based on size and structure. He wrote that "A monolithic approach to systems and programming project management is unlikely to produce the most satisfactory results. There are critical differences in project composition… which influences the mix of tools that should be brought on by its management" (McFarlan, 1974).

Other research efforts have sought an even higher degree of specificity to distinguish differences among phases in the life of a single project. McKeen argues that the dominant organizing framework for application of system development is the life cycle concept. This concept divides the total development cycle into identifiable stages, where each stage represents a distinct activity. This activity is characterized by a starting point, an ending point, and deliverables according to expressed purposes (McKeen, 1981).

The life cycle model was formally acknowledged as an important element in systems development by its inclusion to the information system curricular proposed by the ACM Curriculum Committee on Computer Education Management. Davis argues that application systems all need to undergo a similar process when they are conceived, developed and implemented. If any portion of the lifecycle is neglected there may be serious consequences. The strength behind the concept of a life-cycle model lies in the creative nature of software development. Information systems development involves creativity, and the life-cycle model is a way to obtain a more disciplined creativity by giving the process a structure. The life cycle is therefore important in planning, management and control of software development (Davis, 1974).

Glass provides a common breakdown of the life cycle categorization. (Glass, 1979) He identifies 5 major categories: 1 Requirements and specifications, Design, Implementations in concert with an express purpose, Check out and Maintenance.

The life cycle does not proceed linearly; rather it is an iterative process. For example may review after the system design phase result in going back to the beginning in order to prepare a new design. Boehm's waterfall emphasizes the highly iterative nature of software development (Boehm, 1981).

<u>2.2 Project management and system dynamics</u>

The first part of recent articles I present, concern project management and system dynamics. These papers establish the position of Abdel-Hamid's model as part of the modeling tradition.

In addition, the problems addressed in these publications are all related to the subsystems of the main model, and the key challenges identified. For example, research on delivery times, the 90% syndrome problem and resource allocation.

Lyneis and Ford argue that one of the most successful areas for the application of system dynamics has been project management. They draw on a measure of new system dynamics theory, new and improved model structures, number of applications and practitioners among other indicators to underline this argument (Lyneis & Ford, 2007). Therefore we find a rich source of literature that deploys and utilizes system dynamics on different aspects of management. Some are related to software development directly, while others focus on industries and management in general.

In Homer, Sterman, Greenwood and Perkola's paper published in 1993 for the System Dynamics 93 conference, the authors construct a System Dynamics model in order to reduce delivery times in projects involving engineering, procurement and construction of complex equipment systems for pulp paper mills. The model incorporated features that where original at the time, like a critical path determined by "gates" connecting sequential activities. The model helped the company reduce delivery times with 30%. In the development of the EPC model, the authors drew on the long history of work on this area, including Roberts and Abdel-Hamid. The base for Abdel-Hamid's base model is linked to the same literature, and is now part of the base for modeling (Homer, Sterman, Greenwod and Perkola, 1993: pp.212-216).

David Ford and John Sterman's paper consider the 90% syndrome, and how to overcome it by Iteration Management. The concept of the 90% syndrome was identified by researchers like Baber (1982), and is also an important assumption for Abdel-Hamid's base model (Abdel-Hamid, 1991: pp.198-202). The 90% syndrome is identified by Ford and Sterman as a schedule failure in concurrent development. By increasing concurrence and common managerial responses to schedule pressure they aggravate the syndrome and degrade schedule performance (Ford & Sterman, 2003: pp.177-185).

Jogeklar and Ford's article published in 2003 focus on resource allocation. They argue that shortening a project's duration is critical to the product development project success in many different industries. As a primary driver of progress and an effective management tool, resource allocation among development activities can strongly influence this duration. The author's provide a resource allocation policy matrix as a means of describing resource allocation policies in dynamic systems (Jogeklar & Ford, 2003: pp.72-87).

Bayer and Gann's paper on bidding strategies and workload dynamics construct a system dynamics software model in order to investigate resource deployment and interaction between business and project processes. A common challenge is to manage resources across a fluctuating workload. Firms often accept these conditions believing that there is little they can do in order to moderate these fluctuations (Bayer & Gann, 2006: pp.185-211).

Reichelt and Lyneis paper from 1999 focus on the dynamics of project performance. Research, design and development projects are notorious for failing to achieve cost and schedule budgets. They argue that despite an increase in effort over the years to improve project management, they fail due to model-tools that are not considering the complexity and dynamic nature of such projects (Reichelt & Lyneis, 1999: pp.135-150).

2.3 Software project management and system dynamics

In the field of software development and management, several papers utilizing system dynamics has been published over the last two decades. They provide different approaches to improve software project management. They establish the position of the development cycle, and challenges to management. The range of papers provides two patterns within the software production literature. The first pattern regards modeling foundation. The papers provided here all have the same scientific foundation. From the classic literature they draw on the works of Roberts, Glass, and Abdel-Hamid's system dynamics model.

The second pattern is scope and depth. The different papers start with the same foundation, but differ in scope. Some papers focus on single isolated aspects of software management, while other seeks to analyze the entire phenomenon of software development across time and space.

Lehman and Ramil's paper from 1999 focus on the impact on feedback in the global software process. The authors argue that there is a lack of major achievements in the software industry when it comes to production and maintenance. The paper introduces the Feedback, "Evolution and Software Technology" (FEAST) hypothesis. This hypothesis states that problems stem to some degree from the feedback nature of that process, and such explanations are overlooked in seeking improvement. The paper explores this phenomenon through identifying, modeling and simulating the evolutionary behavior of several industry software projects. The model demonstrates the presence and impact of feedback on process behavior and improvement. From this model the authors provide guidelines for software process planning and management (Lehman & Ramil, 1999).

Kahen, Lehman, Ramil and Wernick published in 2001 an additional paper on the FEAST research, focusing on the consequences of long-term evolution on software production. The authors present elements from previous works, combined with new approaches to investigate the consequences of long-term evolution. This allows an extension of previous models, and identifies the consequences of decisions made by managers in the software process. The model investigates and identifies progressive and anti-regressive activities, and the decisions on the fraction of work that is put into such activities. One example of a progressive activity is complexity control (Kahen, Lehman, Ramil, Wernick, 2001).

Rus, Collofello and Lakey's article from 1999 describes the use of a process simulator to support software project planning and management. The focus of this model is mainly software reliability, but the authors argue that one can transfer the model to other software quality factors. The process simulator was constructed as part of a decision support system for project managers in planning. The simulator was built using the system dynamics approach, and later provided extended strength through the concepts of discrete event modeling. The continuous model can be used in project planning, and as a predictor for the effect from management and reliability engineering decisions. This can also be applied as a training-tool for project managers (Rus, Collofello and Lakey, 1999).

Williford and Chang published in 1999 a system dynamics approach to strategic IT planning. The authors built a macro-scale model on the FedEx IT division. The system dynamics model predicts staffing, training and infrastructure funding over a 5 year period. The model was built using regression on business, system and productivity metrics linked to business projections. The model generated expected demands for development, support and rollout, and management was provided with a dashboard of strategic factors. These factors could be adjusted to determine the impact on workloads and productivity. The exercise at FedEx established IT management's commitment to process improvements, architecture and attention on human resource strategies (Williford and Chang, 1999).

The publications listed above establish the position of system dynamics in the field of software development. The challenges and concepts observed in project management are strongly related to the software project management process as well. The papers show how the scope and width of this topic varies across the discipline. The first two papers seek to expand the understanding of software evolution and production beyond the boundaries of previous efforts. Unlike Abdel-Hamid, the authors here focus on the long-term evolutionary effect of software production on new projects. This provides a wider scope in both space and time. The latter two papers address central challenges for management within the same scope as Abdel-Hamid's original model. They isolate an area of investigation, but provide information and predictions for the product process as a whole. These publications all stress the importance of management and decision on software production and quality.

2.4 The concept, significance and crisis of quality within production and software production

In my research I wish to address the need for increased focus on the software quality aspect of software development. As I argued in the introduction, the current software crisis hinges on the failure to provide accurate estimations on the quality assurance, rework and testing effort as well as planning deficits.

This part of my literature review will address the concept of quality for process and end product and the necessity for it. At the same time I wish to reveal how the concept of quality is essential for both the end-product itself, and the development process. The literature I apply here will focus on product quality and quality of process in order to establish it as a management issue. This is essential for my interest and desire to enhance the existing model using System Dynamics.

The software industry is a relatively young industry. In order to establish it in the realm of production and development it is therefore necessary to look into other more established industries to identify key factors and challenges. Abdel-Hamid drew on experiences from other fields such as research and development to find his building blocks for the software production model. In order to establish the importance of quality I will draw on the classical publications on quality written during the three waves of quality thinking.

The idea of software quality and process quality arrives from the publications on quality in the general industry. I argue that the ideas and observations made in these publications provide insight into the issue that is fundamental for our understanding of quality and ability to develop the model. Software quality and process quality arrives from the general industry through different management tools. One example is the concept of Total Quality management (TQM).

The TQM is a concept of management philosophy where leadership at the top is essential to provide a guiding line for directions and strategies. The concept of TQM is general, and its focal point is the causality between quality and cost-efficiency in industrial environments. Over the latter years, several industries have tried to implement the TQM philosophy and to some degree they have been successful (Sterman, 2003). From the software development industry the quality standards of ISO 9000:2000 is based on the TQM philosophy (Hjertø, 2003: pp.48-51).

The introduction of the TQM at the end of the 1980's is a culmination of contribution from essential scholars from the 1950s to the 1980s. There are in general three waves of quality thinking behind the concept of TQM, and they manifest themselves in the early 50s in the US, the late 50s in Japan and the late 70's by a new wave of American and European scholars. I will now present some essential ideas from these periods in order to link them to my endeavor to enhance and analyze the quality aspect of software development through Abdel-Hamid's model.

2.4.1 The American wave

The American wave of quality philosophy starts at the early 1950's. The first notable contributor that will be mentioned here is Dr. Deming. Deming's work on quality of production provides 14 important guidelines and 5 deadly diseases that over time can destroy an organization. Through these 14 guidelines, Deming argues that quality is something that can only be achieved through proper planning and as an active approach. It is something that must be incorporated by management, and the value of quality must be taken into consideration. Long term gains are more important than short-term profits. Secondly, an organization should always be trying to improve itself and the quality of production. It is essential in Deming's approach to educate and train personnel in order to achieve quality over time, and by learning overheads and disruptions can be nullified (Deming, 1982).

The five deadly diseases of Deming provide the bad traits that will over time destroy an organization. Lack of continuous obligations is the first illness. The second illness considers short term profits. If an organization only focus on short term gains the focus is on profit rather than quality. This in turn sacrifices quality measures in order to provide cost-efficient short term solutions. One can easily draw the parallel here to the QA sector. Abdel-Hamid identifies that a software project behind schedule often reduce the QA effort for a long time-period or indefinitely until testing in order to increase production (Abdel-Hamid, 1991: p.69). If such a practice has been established it's hard to turn it around (Deming, 1982).

The third disease is awards and ranking of employees and performance. This award system creates an "us and them"-feeling between employees were fear of underachieving create increase pressure. Such pressures increase anxiety-levels for the staff, and motivation decreases (Cougar & Zawacki, 1980).

The fourth disease considers turnovers. If managers tend to move between jobs it creates instability in an organization. Management should identify them with the organization as a whole. This way they hold a clear accountability line to the end product as well.

The fifth disease is a warning to managers who only focus on numbers. Numbers can always be manipulated or misguiding. It is important that leadership base their decisions on scientific data, numbers gathered using sound statistical methods (Deming, 1982).

Dr Juran's work on quality and production culminated in his book published in 1951 called "Quality Control Handbook". Juran argues that quality must be planned ahead in order to be fully implemented. For Juran, a firm or producer should make new plans every year in order to reduce costs and increase quality. In order to plan quality, one must build a structure that can ensure quality. This is managerial responsibility, and they must build this structure.

This must be achieved through four steps. (1) Decide on actual goals and milestones. (2) Create realistic plans for milestone achievements. (3) Divide the responsibility between different segments in order to achieve the goals. (4) Reward the organization based on the milestone achievements, while learning from the process (Juran, 1988). Juran notes that 85% of all errors occur due to managerial or systemic errors. The remaining 15% of mistakes are made by workers. Unfortunately the focus from management has been to look for blame and personal mistakes among the workers, rather than addressing the managerial or systemic problem that facilitated the error (Juran, 1988).

Feigenbaum's doctoral dissertation introduced his ideas on Quality and the concept of Total Quality Control. For Feigenbaum it's not enough to ensure quality through production alone, but through management as well. In TQC the focus is relations within the organization and its actors. It is a managerial duty to set standards for quality. Investigate how the organization adapts to these standards and adhere to them. If the standards are not followed management must intervene and also plan to improve the standards (Feigenbaum, 1991).

2.4.2 The Japanese wave

The Japanese way of implying quality and quality assurance builds on the early American ideas only transferred into Japanese production culture. This culture is slightly different from the American culture of production as it is process oriented rather than only innovation oriented.

Quality should be achieved through a system of quality. This concept is named Kaizen, which means improvement. The idea behind Kaizen was first introduced by Masaaki Imai, and focus on a quality structure that capsules both product and production process. Small improvements require small investments, but over time they provide robust and foreseeable production environments (Imai, 1986). There are three scholars that are central to the Kaizen philosophy: Ishikawa, Taguchi and Shingo.

Ishikawa's contribution to Kaizen focuses on quality circles. These circles consist of employees that meet on a voluntary basis. The circles should provide ideas for improvement and evolution of the organization. It should develop cooperation between employees and make it more enjoyable to be at the workplace and enable management to utilize the peak efficiency of every single individual.

The idea of quality circles and voluntary participation is to provide positive motivational factors regarding quality. This is very important in software quality as well. Rework is the processes where bugs and errors are fixed after the quality assurance effort has detected them. This is an area where such positive motivational factors are important. Rework is done by the production team and involves recoding a piece of the software. This effort can be regarded by developers as a "chore" or "punishment" for mistakes.

When we look at reasons for bad fixes motivational factors play an important role. If the developer is stressed, under pressure or lacks motivation the chances for bad fixes increase (Brooks, 1982). Ishikawa's idea is to provide positive group motivation for quality processes like rework. If rework is not regarded as a chore but an opportunity for improvement, the motivation to perform this task increases.

Taguchi's contribution to Kaizen is coined the "quality loss function". Losses are imposed on the organization as work has to be redone, be scrapped, or go through additional maintenance. This is supported in the software literature by both Charette and Jones & Bonsignour (Charette, 2010) (Jones & Bonsignour, 2012).

Losses from bad fixes, delays or poor quality and functionality create double losses for the producer, in terms of lost market shares due to low quality or customer satisfaction. In order to avoid such a situation one should provide parameters of quality, and immediately intervene from management's side if the current product is deviating from the quality parameters. Granted this concept is hard to translate into a R&D or Software setting as they are hard to determine with set parameters due to their creative and design-heavy starting point (Taguchi, 1981).

Shingo's contribution considers the flow of production between phases when errors occur. For Shingo management should halt production when an error has been detected. This should be done in order to identify its cause and repair it before the error can cause damage to the process. This way one can identify where the sources of error appears, and create preemptive measures. Shingo’s philosophy is called poka-yoke which translates to “no errors” (Shingo, 1959).

2.4.3 The new wave

The new wave contains American and European scholars who are focusing on quality in the late 80s (Hjertø, 2003: p.65). Phillip B Crosby’s work on quality stems from his time as vice-president of ITT and later as an independent consultant for management and leadership. Crosby’s argument rests on the principles of Deming, and constitutes four main absolutes.

Quality must be defined, and given proper parameters. The quality control system must be designed to detect and prevent errors. The sought error-rate must be equal to zero. Even if it is impossible to detect and remove all errors, as argued by Abdel-Hamid in software development, the goal must be zero. If one starts to adapt standards that allow “some errors” the incentive to track and detect errors diminishes.

Finally quality must be presented in financial terms. Waste, rework and delays should be clearly identified as costs, and given an actual price tag. This in turn would provide more reliable cost-estimates from the software industry as the costs of rework or delay would be calculable (Crosby, 1979).

For Crosby the idea of quality must be incorporated into the organization as a whole. Quality is not something one achieves by chance, but by active management and experience. This notion is supported by Claus Möller who argues that quality is achieved in an organization were quality is incorporated in the organization as a whole by management (Crosby, 1979).

When we look into the principles applied from the different time-periods on the concept of quality in production we find some interesting patterns. There are observations on problems regarding quality that we can trace into the software industry itself. First of all, the question of quality is a management issue. 80% of problems regarding lack of quality stems from lack of proper management. According to the authors there is a quality crisis in the western world that continuously creates losses and user dissatisfaction. There is a clear link between a proper process of production and quality. There is a constant need for improvement in the industrial processes, and one need to address and prevent failures at an early stage rather than after the product has been finished (Hjertø, 2003: p.59).

2.4.4 The client and quality of product

From the early waves of development, we find a focus on the customer/client in the question of quality as well. When we look at the fundamental philosophy behind the approach of both Juran and Feigenbaum in the first wave, the client is the important factor. Juran stress the importance of knowing the customer. In order to provide true quality of the product, it must be constructed according to the wishes of the end-user (Juran, 1968). This notion is also apparent in Feigenbaum's philosophy. For Feigenbaum, the quality of the product is a way to promote business. In order to ensure this quality, satisfying the customer is of vital importance. It is important to build a relationship with the client in order to understand his needs, and how to embed them in the product. This is built by human relations inside the organization and between organization and clients (Feigenbaum, 1980).

The focus on client and customer satisfaction is at the main pillar for the more recent movements in the third wave of quality philosophy. Philip Crosby argues that the first obligation for an organization is to satisfy the customers and clients. This must be achieved through an active quality system containing both activities and policies (Crosby, 1979). Crosby points at measurements and registration of quality as factors for success. The production of a commodity must adhere to the client's needs and standards. If the end-product violates these standards, the quality becomes mute. If the product does not reflect the client's demands or needs it is a failure.

Tom Peters and Robert H Waterman's philosophy holds two cornerstones: Continuous innovation, and exceptional sensitivity towards the client. For Peters the clients are both external and internal. External clients are customers that should experience great service and exceptional product-quality. Internal clients are members of the organization, and should experience excellent leadership (Peters & Waterman, 1982).

From these philosophies we see that a fundamental focus on the client is apparent from both the 50s and the 80s. In the software development literature, we also find the client to be an important aspect of software development. The client is the end-user of the product, and therefore holds a key position in regards to software quality. From the description of the software crisis, we see that poor client satisfaction continue to plague the industry as a whole (Charette, 2010).

Hjertø argues that it is important to form and maintain a productive relationship between the software production-side and the client-side. The client holds a lot of information about the needs and standards the end-product must adhere to. To produce software to one specific client in order to solve or cover a very specific need, places software development in the realm of bespoken production. The needs and standards may vary between each project, and the development of a default standard of including the client is vital. If the client is incorporated in both planning and end-user testing, the needs and ultimately the quality will benefit the organization (Hjertø, 2007).

From the data collected by Jones and Bonsignour on the software development industry in the US, we observe a pattern of allowing the client to participate at a higher degree than earlier days of software development. With the progress of software allowing easier user access, the opportunity for development actors to incorporate clients for testing and fine-tuning has increased as well (Jones & Bonsignour, 2012). The result is a host of different testing-forms that involve users or clients. Therefore the client has been transformed into an active participant rather than a passive source of information. Among the forms of client testing we find schemes such as Usability tests, Lab Testing, Field Beta Testing and Customer Acceptance testing. The authors demonstrate that each kind of test allows additional errors or weaknesses to be found. This information enables the production team to remove more defects and provide better software quality.

DeMarco and Lister provide a counter-argument to including the client in all aspects of software development. Their argument rests on the technical nature of software and software development. The client may provide ideas and standards that are important, but the professional designers and software producers hold more knowledge. If software production is reduced to a process only catering the client’s expressed needs, the room for innovation is reduced considerably. This will eventually lead to demotivation and loss of pride in the software development staff. If the question is quality, then the client should be an integrated part. If the question is “which solution is better?” the experienced personnel should have the mandate to make the choice (DeMarco & Lister, 1999).

From the literature it is apparent that the client is an important factor to incorporate for software quality. The increase in client review and customer acceptance review indicate that the industry have established such channels.

2.4.5 Quality and software production and its crisis

From the different waves of quality-philosophy we see a pattern emerging that can easily be translated into the current software crisis. If we look at reports like the Standish Report, and the ESSU report, it becomes apparent that the software production environment has severe challenges when it comes to costs and quality. The fail rates are high, and the costs of rework, and post-maintenance substantial (Standish report, 2005) (ESSU, 2007). There are several empirical examples as well that highlight the lack of software quality and the consequences of them.

One example we find from the US. In August of 2006, the U.S. government, student loan service erroneously made public the personal data of as many as 21,000 borrowers on its web site, due to a software error. The bug was fixed and the government department subsequently offered to arrange for free credit monitoring services for those affected.

A second example following the same vain concerns a software error reportedly resulted in overbilling of up to several thousand dollars to each of 11,000 customers of a major telecommunications company in June of 2006. It was reported that the software bug was fixed within days, but that correcting the billing errors would take much longer. *(www.softwareqatest.com)*

Such examples as the one we find from the US produces lack of confidence in software from the customer. They are a product of poor quality, and generate huge losses. The ESSU report from the European Union support the US examples as it examines the trend in 105 outsourced governmental IT projects. In December of 2007 Dexter Whitfield at the European Services Strategy Unit (ESSU) published a report on outsourced public sector IT projects. This report's and title even echoes that of the Congressional report of 1979. The ESSU report is named: "Cost overruns, Delays and Terminations: 105 outsourced public sector ICT projects" (ESSU, 2007).

The ESSU report shares many of the findings reported over the last decades. The ICT sector is hallmarked by delays, cost overruns and implementation problems. The total value of the contracts was £29.5 billion, while the total cost overruns totaled an additional £9 billion. 57% of all contracts experienced cost overruns at an average of 30.5%. In 33% of all the 105 programs one experienced major delays, and 30% were terminated (ESSU, 2007).

The ESSU reports identify several areas that cause the costs to sore due to underestimation. These are linked to additional staffing due to technical issues or consolatory staff as the project passes through the different production stages. The report also concludes that the problems are caused by managerial shortcomings. In the NAO's report from 2003 these traits are summarized and used in the report as reference. In this list several key problems include statements such as:

1. Lack of a clear link between the project and the organization's key strategic priorities including agreed measures of success.
2. Lack of senior management and ministerial ownership
3. Lack of skills and proven approach to project management and risk management
4. Too little attention to breaking development and implementation into manageable steps.
   (ESSU, 2007)

These hallmarks identified by the ESSU fits together with Charette's reported common factors for software failure. His hallmarks include for example unrealistic or unarticulated project goals, inaccurate estimates of needed resources, badly defined system requirements and sloppy development practices. Both these publications from the end of the 2000's portray an image of the software industry as an industry in a crisis (Charette, 2000).

From the ESSU report we find several examples of failed projects due to poor quality and sloppy processes. The first example we draw on comes from the British department of work and pensions. In a project with IBM, Seibel and Curam the design and implementation of a new software program to streamline benefits processing was contracted. The project was terminated in August 2006 after 3 years of production. The end product was never used despite the fact it was based on an off-the-shelf product.

The second example is drawn from the Child Support Agency (CSA). The CSA contracted the EDS to create a new IT system for its operations. This project was terminated in 2006 due to poor quality and overruns. The report stated that “The new IT system performed no better than its predecessor. Systemic problems, over 40 internal audit reviews and an additional £91 million caused the project to be terminated. Costs soared out of control.

Both examples from the ESSU report illustrate a lack of quality problem. With new templates and even a base-product to build the new software on they experienced overruns delays and poor quality. Both projects were terminated right before finish or right after delivery (ESSU, 2007).

In my introduction I mentioned an example from Norway regarding the Altinn governmental portal. This example is important as it highlights the importance of testing in a software project, and the consequences of poor testing-routines. In March 2012 the governmental portal Altinn (produced by Accenture) which functions as a delivery-post for official documents such as tax-records broke down due to software errors. For several hours any person who logged on to the portal, got full access of the entire records for one single individual. These are classified documents, and such an error gravely injures the relationship and trust between user and government.

The Altinn portal had to be taken down, and could not operate for several weeks. The independent Norwegian agency for control and verification named Det Norske Veritas (DNV) started their investigation into the problem. From their report it is clear that poor software and testing caused the problem. The software holds 3500 significant errors that are now under rework. The report also concludes that “Quality in testing has been poor in all stages of production. The existing test regime is inadequate and DNV cannot verify that the established crisis-management regime will be sufficient for future problems” (DNV, 2012: p.4).

Lack of testing led to a snowball effect, where problems that should have been detected earlier in production slipped through. They manifested themselves in the following production stage. The lack of a clear test regime made rework harder as certain key tests was not conducted, and more errors occurred between the stages of production as a result (DNV 2012: pp.30-35).

Jones and Bonsignour identify the observed empirical pattern of low quality in their book on the economics of software quality. They argue that the software industry has the highest fail-rates than any other industry. The failures often hold dire consequences, as software is an important part of our integrated systems. When a car's breaks fail, or a hospital make a medical mistake it's often caused by software failure (Jones & Bonsignour, 2012: pp.4-6).

The second pattern that is derived from the software crisis is the fact that it is a managerial and a system problem. Just as with quality of production, the need for firm management and plans to achieve end-quality is needed (Sterman, 2003). This follows a process oriented approach to quality, which is vital in a system dynamic effort. This falls under the Technical or structural quality definition, and the process quality definition identified by Jones and Bonsignour (Jones & Bonsignour, 2012: p.11).

2.5 System Dynamics and Quality

From the System Dynamics literature on the field, there are several articles addressing the question of quality and the challenges to it. Repenning and Sterman's article "Creating and Sustaining Process Improvement" from 2001 argues that the even though managers have more tools available to learn about different improvement tools, they fail to implement them. Quality and improvement of process is not something that can be created as a simple solution. They observe that the inability of most organizations to reap full benefit of innovations has little to do with the specific improvement tool rather how it interacts with the existing environment. Hence, the problem is systemic and not isolated to human resources, or leadership (Repenning & Sterman, 2001).

In Black and Repenning's article on the nature of rework and resource allocation the same pattern of an increased need for planning ahead in order to sustain quality is observed. Under-allocation of resources early in a project persists, and causes increased rework and pressure as the project reaches its end. The article observes that their policies fail to improve the situation, as it requires an overall strategy in order to solve the issue (Black & Repenning, 2001).

2.6 Summary

In this literature review I have established the position of Abdel-Hamid’s model in the field of system dynamics, through the literature it is related with. Secondly I have provided a sample of literature that has been published after 1991 in order to show how the concepts of project management and software project management and their issues portrayed still remain essential in the System Dynamics literature. Thirdly I have established the concept, significant and crises of quality and related it to software development and software end product. I find common traits such as the need for increased planning and quality awareness. The concept of quality is much more than a question of end-product, it has to be established through quality of process. I also discovered that in the system dynamics literature, this concept has not yet been thoroughly examined related to the base model.

Chapter 3: The system dynamics method

3.1 Research method

Through the software production literature, it becomes quickly apparent that the production environment is highly complex. This is one of the reasons management of software development has proven to be such a challenge over the last 40 years. In order to improve our understanding of the mechanisms and provide policies, we need to deploy a method that allows us to model and understand the behaviour of the system. The method must be able to portray complex systems, and simulate the relationship between variables across time and space.

The System Dynamic approach is a method that focuses on just the above mentioned necessity. The approach seeks to provide understanding of complex dynamic systems over time. This is achieved through the concept of internal feed-back loops and time-delays that will influence behaviour in the system as a whole. For Sterman, System Dynamics as a method facilitates learning inside complex and non-linear systems, where the concept of both feed-backs and time-delays create misperceptions. In System Dynamics such misperceptions can be identified and corrected if we have correctly calculated and represented key factors and behaviours inside the system itself. Hence, the system dynamics approach allows us to build and test policies and assumptions in order to improve understanding of system behaviour or to change the observed behaviour (Sterman, 2003).

There are three building blocks in System Dynamics that are essential for modelling behaviour and providing policies namely feedback and causal loops, stock and flows and delays.

3.1.1 Feedback and Causal loops

Feedback is essential in system dynamics. Much of the art in this method is the ability to discover and represent feedback processes. All dynamics arises from the interaction of feedback. There are two kinds of feedback loops in the world; they are either positive or negative feedback loops. The positive feedback loops reinforces or amplify the processes that are apparent in the system. For example a bacterial growth system provides a positive feedback system. Bacteria will multiply over time, and the more bacteria that is present in the system the more bacteria will spawn. This process can in theory run endlessly.

Negative feedback loops are self-correcting systems that counter act change. One example of a balancing system is a classic lotka-volterra system. The numbers of hares that can survive depends on the amount of food in the system. If the population of hares increases above the sustainable level of the food-supply, the overpopulation will starve to death. This feedback then reduces the hare-population to its former level over time. It is vital to understand the interaction between these kinds of feedbacks. The complexity of a system is not derived from the amount of variables or complexity of structures, but rather from the feedback relationships between components and variables of the system (Sterman, 2003).

Since feedback is such a core concept of system dynamics, one of the first tasks for a modeller is to capture and track feedback in the observed system. This is done by using the concept of "Causal loop Diagrams" (CLD). CLD's have a long tradition in academic work, and increasingly common in for example business environments. CLDs are excellent for capturing hypothesis about causes of dynamics. It may be used to elicit and capture mental models or communicating important feedbacks responsible for a problem. A causal diagram consists of variables connected by arrows denoting the causal influences among variables. The important feedback loops are also identified in such a diagram, and variables are related by causal links. The pictures below portray the different components of a causal loops diagram (CLD).

Figure 1. Positive and negative relations

In the figure above we see two kind of arrows used in a causal loop diagram to portray the relationship between two variables. The polarity is given by the + or – sign at the arrowhead. When there is a positive relationship between two variables, an increase in one variable provides an increase in the other. If there is a negative relationship between the variables, an increase in one provides a decrease in the other. The little CLD provided below shows a causal loop between two variables.

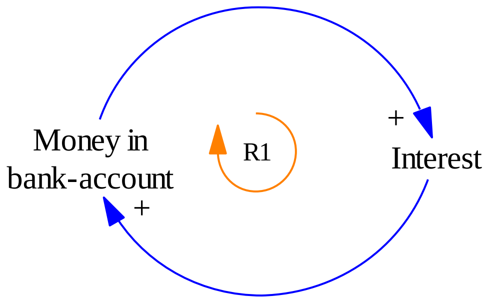


Figure 2. Positive (reinforcing) feedback loop

This small CLD is a simplified depiction of money in the bank-account, and interest. When there is money in the account, the bank will pay interest over time. The more money in the account, the more money is paid in interest. Therefore the relationship between the two variables is positive, and this causes a reinforcing loop. This is provided by the R1 sign with an clock-wise arrow around it. For reinforcing loops the arrow is clock-wise, while for balancing loops the notation is a B with a counter-clockwise arrow. The CLD below shows a complete causal loop diagram with both a reinforcing and a balancing loop.

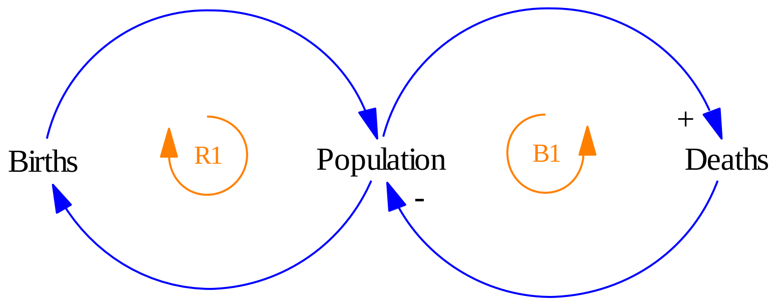


Figure 3. Positive (reinforcing) and negative (balancing) feedback loops

The figure shows the relationship between the total population and births and deaths. When the population increases, the number of fertile humans increase and more births will occur. With more births the population increases. When the population increases, more humans reach old age and will die. Increased population leads to an increased death-rate, and the overall population will decrease. This little system provides the reinforcing and balancing aspects of a CLD, and shows how two loops are causing the behaviour in the population-stock.

3.1.2 Stock and Flows

The second core concept of System Dynamics modelling is stock and flow. Causal loop diagrams are useful in many situations. However, they have their limitations. The most important limitation is the inability to capture stock and flow. Stocks are accumulations, and they characterize the state of the system. Stocks generate information that decisions and actions are based upon. Stocks provide a system with both memory and inertia, and are the source for delay. Stocks provide this delay by accumulating the difference between inflow and outflow to a process. Through decoupling rates of flow, stocks are a source of disequilibrium in a system (Sterman, 2003). One example of a stock and flow system is a firm's inventory. The inventory increases by the flow of production and decreases by the flow of shipments. Another basic example is the bath-metaphor. The tub is the stock and the water-level increases by the inflow of water from the tap, and the outflow of the tub is the drain. The tub-metaphor also provides an example of delay. A water-filled tub will be drained from water if the tap is turned off and the drain opened. However, this does not happen instantaneously as the water will leave the tub in a smooth fashion over time (Sterman, 2003).

The picture below is taken from Sterman's book "Business Dynamics" and depicts a small stock and flow diagram. The stocks are depicted as squares, while the inflow and outflow are depicted as pipes entering or leaving the system.

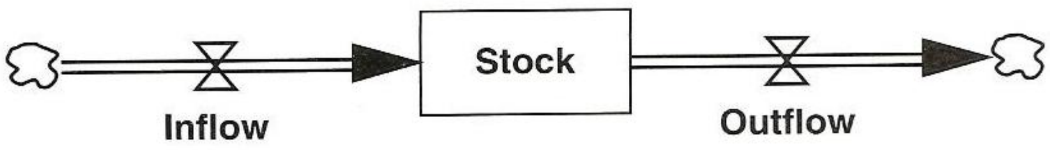


Figure 4. Simple stock and flow diagram (SFD),
Source: Sterman (2003, p.193)

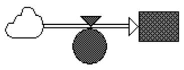
The stock of the system is depicted in the square, while the pipes with valves control the inflow and outflow. The clouds at beginning and end of inflow/outflow depict variables or processes outside the model boundary.

In a system dynamics model, there are three kinds of variables that are utilized when a system is constructed and simulated. These will all be present in a stock and flow diagram.

(1) A constant is a variable that is initialized at the first time-step of a model and kept constant during the simulation run. The symbol for the constant is a diamond-square ◆.

(2) The stocks of a system are also initialized at the beginning for a simulation. Unlike the constants, stock accumulates its value over the model run. The operating inflows and outflows will govern this value. The symbol for a stock is the square as depicted in the previous figure.

▩

(3) The final variable in a stock and flow model is the auxiliary variable. These variables calculate information and forward it through the system. The auxiliary variables are calculated by each time-step through the model run. Auxiliaries are used to control the flow-rates of a stock, and are depicted as circle-symbol: ● When the auxiliary is connected to a flow-rate the symbol is as follows: . Flow rates govern the input to, and output from stocks (Sterman, 2003: pp.193-195).

3.1.3 Delays

Delays are an important source of dynamics in all systems. Some delays create dangerous effects by creating instability or oscillations, while others provide clarity by filtering out unwanted variability and enabling managers to separate signals from noise. Delay is a process whose output lags behind its input. One example is the process in which a letter is being sent from sender to recipient. The time it takes between the letter has been put in the letter box until it arrives in the box of the recipient demonstrates how letters in transit constitutes a delay. Measuring and reporting the result takes time, decision making takes time, and it takes time before decisions enter into effect in the system. Therefore it is crucial to understand how delays behave and how to represent them. What kind of delays that exist, and how they influence the system that is analyzed.

These concepts are all crucial in my analysis of the software development process. My research demonstrates how the concept of delay and causal relationships provide complex interactions between key factors.

3.1.4 Nonlinear relationship

In our mental models, we often understand the relationship between two processes to be linear. When we double the amount of X, the effect on Y is doubled as well. In the real world however, it is very rare that effects are proportional to the cause over time. This is one of the challenging aspects of understanding the cause and effect relationships around us. What is characteristic for a relationship at one time, may be opposite from its characteristic at another time.

One example we can use to illustrate this is the effect of drugs on healing. When we take a drug to cure an illness, it will have a positive effect on healing. Increase dosage for Medicine (X) and we see an increase in Healing-rate (Y). This effect is interpreted to be linear at first; more medicine yields a constant effect on healing. However, we know that a drug is effective up to a certain point. The increase in dosage will provide more healing, but not as strong as the initial response. Eventually the increase in dosage will only provide a small increase in the healing rate. Depicted in a graph, the line would start with a clear increasing line between X and Y. However, as the effect of increase in X on Y levels off, the line would flatten out. The shape of the graph would be a concave curve rather than a straight line. We also know that over-medicating a patient is dangerous. If the dosage of medicine is above a certain point, it will have a negative effect on the patient, who will experience a degraded healing-rate and eventual death. Now, the relationship provides an “overshoot and collapse” figure, as the harmful effect of overmedication drives the healing-rate downwards.

The software production system holds these kinds of relationships as well. What initially looked like a cure for a problem could eventually end up as a curse. It is very important for management to understand the non-linear relationship between variables. Very often the results from a solution may look counter-intuitive when the effect changes (Sterman, 2000). If the doctor follows the medication logic, more medicine should provide more healing. Instead, more medicine killed the patient.

Building system dynamic models, with both stock and flow diagrams and causal loop diagrams allows us to identify the non-linear relationships over time. The ability to simulate decisions and their effects on the software development process will provide valuable insight for managers. It will allow them to understand the true relationship between effects over time, and when to adjust their approach due to non-linearity. This is the strength for the system dynamics method, and why I choose to deploy it for my investigation of software project management.

Chapter 4: The structure of software development project management

4.0 Introduction

In this chapter, I present my first major contribution to the base model of Abdel-Hamid. In his doctoral thesis, a system dynamics model was constructed in order to experiment with the subsystems of a software development project. However, there are two aids in system dynamics to portray dynamic relationships between variables; stock and flow diagrams and causal loop diagrams. Abdel-Hamid employed only the first method with, stock and flow diagrams. This fact causes a lack of transparency that is causing a fundamental problem for later utilization of his model. The model itself only provides the variable structures inside the model. It does not reveal the important relationships between the variable within the sub-sectors and between subsectors. The important feed-back loops are not available. The result is a situation where researchers, scholars or managers who wish to utilize the model is not able to observe the dynamic relationships inside the model. The reasons behind the observed behaviors when running the simulations are hidden. This is a vital flaw that I will correct in this chapter. My important contribution is a complete set of causal loop diagrams for all the subsectors, and for the model as a whole. This will enable us to examine the dynamics behind the challenges is software development project management, and further develop the logical relationships between variables within and across the subsystems in the model. This is important for my own ability to create enhancements to the model, and it is important for future research utilizing this model. In this chapter I will present the construction of Abdel-Hamid's original model while introducing a causal loop diagram for every structure presented.

4.1 The need for an integrated model

Abdel-Hamid argues that there is a need for an increased understanding of the software development structure as a whole, and the challenges facing managers in a software production environment. Managers make errors, based on their understanding of the software production process. Managers have their own perceptions of mechanism within a project. However, these perceptions are often too simple and therefore unable to connect the different dynamic relationships between processes. Abdel-Hamid provides this figure to illustrate the point. The model depicted below is a simple model of the Software Development Process.

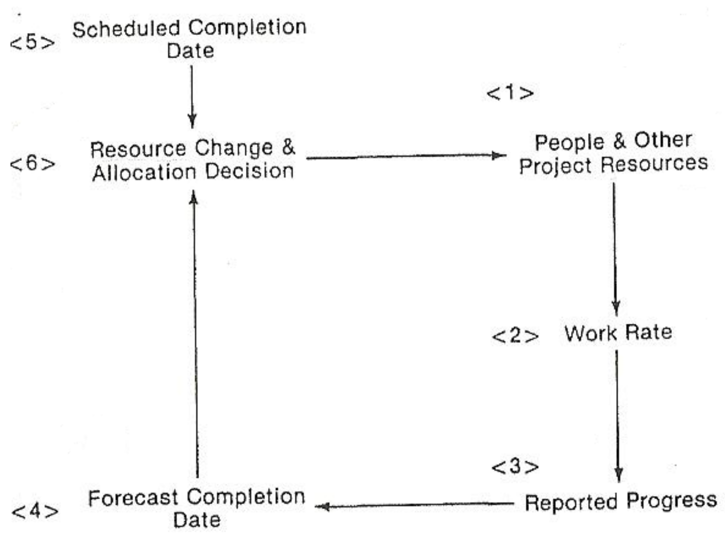

<5> Scheduled Completion Date
<1>
<6> Resource Change & Allocation Decision
People & Other Project Resources
<2> Work Rate
<3>
<4> Forecast Completion Date
Reported Progress

Figure 5. A simple software development process model

This is a simple and tantalizing model that describes how project work is accomplished through the different stages. The allocation of manpower and resources provides the work rate. The work progress is then reported through a control system. The reported progress then leads to a forecasted completion date. The feedback loop is closed as the difference between the forecasted completion date and scheduled completion date causes adjustments in allocation of resources to the project. This is a simplified model that easily portrays the process, however in response to challenges it holds clear weaknesses.

This model suggests for example that there is a direct relationship between adding people or resources and the work-rate. When more people are added the more work is accomplished. This would then suggest that if a project is running behind schedule the solution is to always allocate more resources and manpower. However, this ignores one vital aspect of the dynamic of software projects. When adding additional manpower to a project, it leads to higher communication and training overheads. This in turn dilutes the team’s overall production-rate, and the project falls behind schedule even more. This will delay the project even further. This vicious cycle is often referred to as Brook's law which states "adding more people to a late software project makes it even later."

This example illustrates clearly the need for a more detailed model and that there are many variables both tangible and intangible that affect the software development process. These variables are not independent but interconnected in complex ways (Abdel-Hamid, 1991: pp. 16-18).

4.2 The Base model

In this part I will introduce the core dynamic parts of the base-model. This will constitute a walk around of the model, were the different subsystems are explained and identified. First I present the overall characteristics of each subsystem. Afterwards I present the Causal Loop Diagrams (CLD) that provides the relationship between the different variables inside the subsystem and between subsystems. These relationships are one of my core contributions to the base-model, and will clarify the dynamics across subsystems.

4.2.1 The holistic model of software development project management

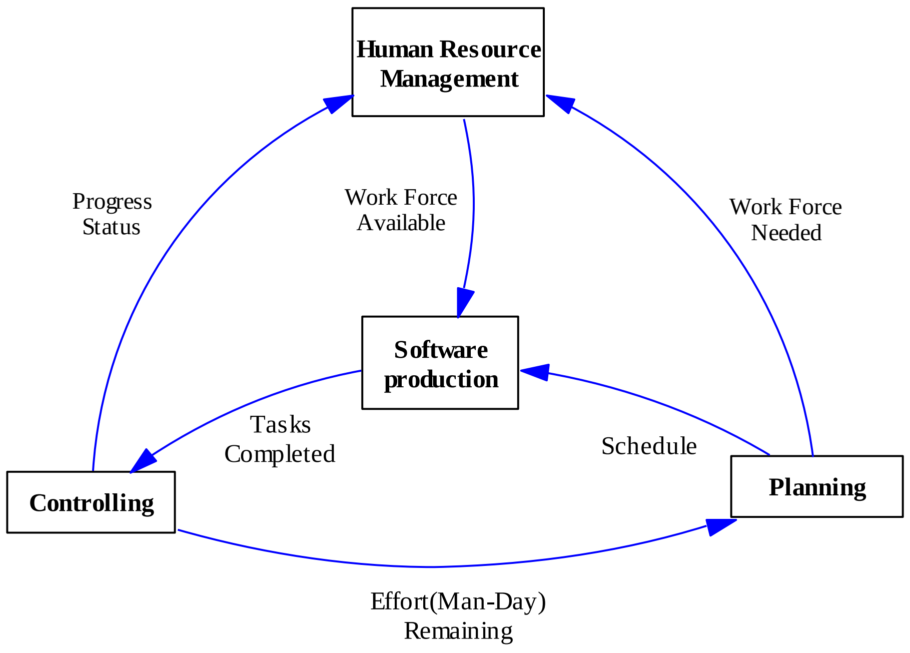


Figure 6. Software development project subsystems

The figure above portrays the software development process as four major subsystems: Human Resource Management, Software Production, Controlling and Planning. This triangular system shows how each subsystem is related to one another through different aspects of the software development process, and provides the starting-point for building the complete dynamic model. This is a simplified depiction, but it is useful as a starting point for building a complete system dynamics model (Abdel-Hamid, 1991: p.18). Through this presentation, these subsystems will be opened step by step in order to provide the key factors and the relationship between them, and provide the rough outline of the model.

When a project has been established with a time-frame, budget frame and functional parameters, the Human Resource side of management allocate workers to the project. The initial workforce is decided by the pre-established parameters through the initial planning effort. It is important to note here that the boundaries of the model exclude this preliminary process. The system starts to operate as soon as the contract has been established and parameters set.

The allocation of workforce to software production allows the production-phase to initialize. Design and coding tasks are carried out by the different production-teams and the results are reported to the Controlling subsystem. In Control, the tasks are subjected to scrutiny-control and quality checks. Based on these checks, Control provides Human Resource Management with progress reports, while an estimation of how much effort remains (measured in man-days) is sent to the Planning subsystem. Based on the estimated effort that remains, Planning provides a schedule for Software Production. During the entire production-effort Planning will continue to reassess the schedule based on the effort that remains, and if needed provide Human Resource Management with workforce estimates. If the workforce is too large it will recommend work-force reduction. If the workforce is not able to finish the product within the schedule, Planning will recommend an increase in workforce.

This system runs dynamically through the software production process were management will control schedule, manpower-allocation and pressures based on reports from Control and Planning. This is an attempt to show the original holistic model with feedback structures. It is apparent that the information provided here is too scarce, and a more detailed model is appropriate.

4.2.2 The simple causal loops of software development project management

On the basis of the first two holistic models and the description of the subsystems provided by the initial thesis, I have constructed the first major CLD for the Software development Project Management system. This first CLD consists of six causal loops spread across the entire software production spectrum. I will now go through each variable step by step for each loop in order to identify the relationships and mechanisms behind them. The figure below depicts the major CLD. This diagram provides a large-scale simple model of the software production system.

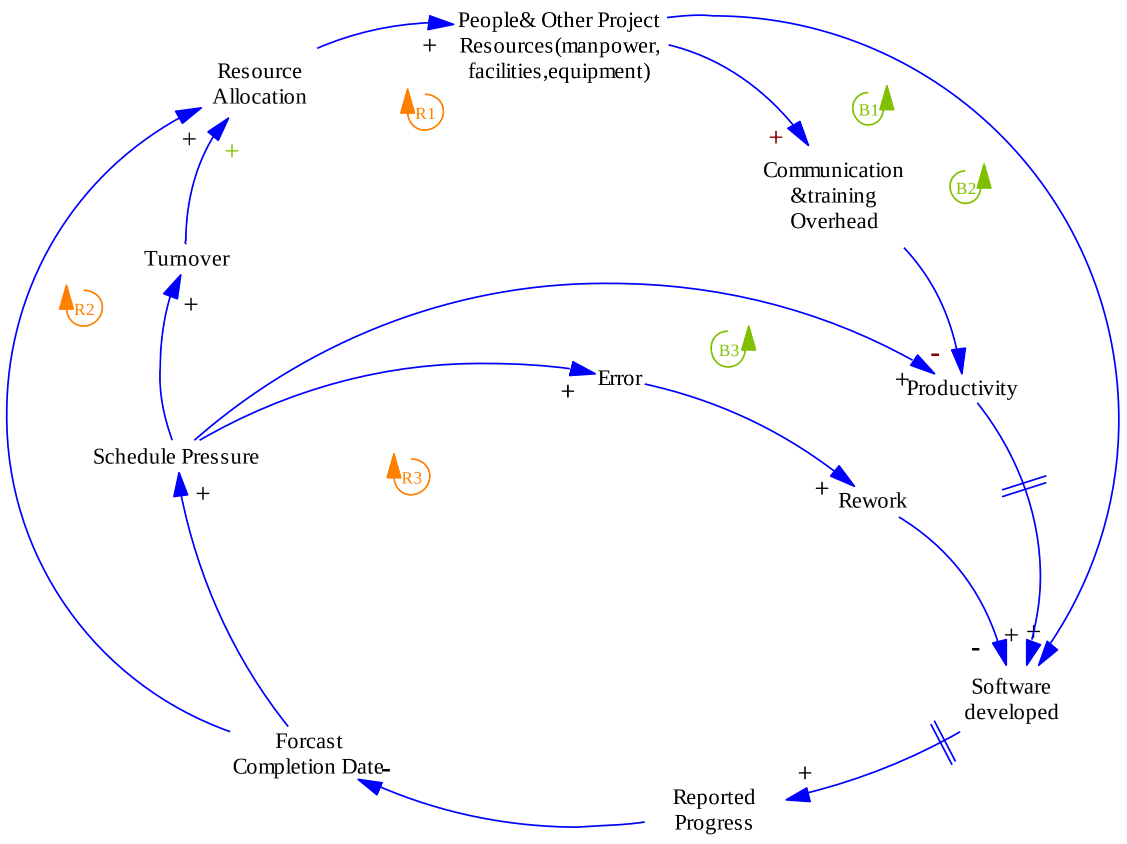


Figure 7. The simple causal loops of software development project management

B1: The first balancing loop in this system focus on the allocation of resources based on reports as the project progress. When the project has been contracted and established, manpower allocates resources to the active project. These resources include pure manpower for staff, facilities for the production and equipment. When resources have been allocated, the project commences and software is developed. During development, tasks are allocated and completed by the production team.

The production team then reports their progress to control after some delay. Progress reports accumulate over time, and are sent to control in planned intervals in order to provide management with the necessary information on progress. The progress reports provide basis for forecasted completion time. Management creates such forecasts on the initial scheduled completion time, and the progress reported. If the progress is less than expected the forecast will give a completion-date later than schedule, and if progress is strong then a completion-date earlier than scheduled is forecasted.

From the forecasts, management is provided with an opportunity to reassess their resource allocation. If the project is running smoothly, no need for major resource adjustments are necessary. If the program runs behind schedule, on the other hand, the need for resource-adjustments will be present. The loop is closed when the forecasts provide a need for adjustments to resource-allocation which causes changes in the allocation of manpower, facilities or equipment.

R1: The first reinforcing loop in this system concerns resource allocation and overheads. As the project moves along, a second effect comes into play. This effect is related to productivity and the ability for the workforce to accomplish tasks. Communication overheads and training is described as a source of delay that dampens productivity. When new people are hired into a project, they need to be trained and integrated into the production-cycle. The only available manpower to train the new hires is veteran workers. This causes a drag on production, as the experienced personnel’s time is now divided by developing software and training new employees.

The second factor of loss to actual production stems from communication. In a software development project, teams must communicate between each other in order to progress. For example, a designer must confer with a coder in order to answer any questions the coder may have on the design-aspects. Both designer and coder must then communicate with the individual testing the code in order to provide their experience with the program. This branched process of communication shows how communication is an overhead. With groups working on a project rather than just one individual, time is spent communicating between members that decrease the production-rate.

The increase in training and communication overheads leads to a reduction in productivity. This reduction will after some delay make itself apparent in the total software developed. When the amount of total software developed is altered due to communication and training, the new forecasts will reflect this loss. When a project is perceived to be behind schedule, the pressure on the work force increases. Many workers allocate more of their time to the job in order to pick up the slack. When pressure is high, the strain on each worker to perform will eventually lead to exhaustion. Working under such conditions over a longer time-period is not beneficial for the workforce per individual or as a whole. Therefore increases in pressure leads to a higher turnover. Turnover is the rate at which workers quit their job and needs to be replaced by management. If the schedule pressure is mismanaged, the end result could be a huge staffing problem. The turnover rate leads to an adjustment for resource allocation, and the loop is closed at the allocation of manpower, facilities and equipment.

B2: The second balancing loop takes into consideration the influx of the forecast and schedule pressure. As the project progresses and software is developed, reported progress provides a forecast for completion. If this forecast indicates that the project is behind schedule, the schedule pressure will increase. Increased schedule pressure will affect the turnover rate, and lead to higher frequency of turnover. Increased turnovers create the need for resource allocation and the loop is closed as adjustments are made by management through the allocation of manpower, facilities and equipment.

B3: The third balancing loop in this CLD concerns the effect of added schedule pressure on productivity. When the progress reports and the forecasts indicate a project running behind schedule, the initial schedule pressure provides an increase in production. When production falls behind schedule, research shows that workers tend to cut their slack time, and devote more concentration to the work at hand. Workers increase their man-hours in order to bring the project back on schedule. This effect increases the actual productivity as workers not only work harder during working hours, but also log overtime hours in order to close the time-gap.

This in turn leads to an increase in productivity that provides new progress estimates. The increase in productivity provides a forecast within schedule and schedule pressure subsequently decreases.

R3: The third reinforcing loop focuses on the schedule pressure and error rate. With increased schedule pressures the workforce will work harder to catch up. However, in a situation with pressure and stress the tendency to make mistakes increase as well. When programmers are tired, stressed or close to deadlines they make hurried decision that lead to more mistakes. With the added production rate they also produce more tasks per hour and may generate more errors per hour as a consequence. With an increase in the errors made the need for rework increases as well. The rework process is a process where faulty code has to be rewritten. People work harder and faster but not smarter. The result is a loop where more errors and rework leads to less production. Less production leads to less progress and forecasts behind schedule. This will in turn lead to more schedule pressure and subsequently more errors committed.

This CLD provides a more detailed explanation than the first holistic models. The relationship between different mechanisms in the system has been provided at a higher degree. However, this is still far from being a detailed model. There are many mechanisms that still need to be explained further in order to provide the needed level of detail. In order to obtain this level, we need to provide CLDs for each subsystems and subsectors one by one. In the end this will be combined into a larger overall model.

4.2.3 The human resource management subsystem

The first subsystem that will be opened is the Human Resource subsystem. This subsystem controls the hiring and management of the total workforce. It is important to make a note here regarding my use of colors in the causal loop diagrams. The variables and relations that are provided in light blue are variables from other subsystems, and causal relations across the subsystems. I will also provide the original subsector for each variable as I address them through the loops.

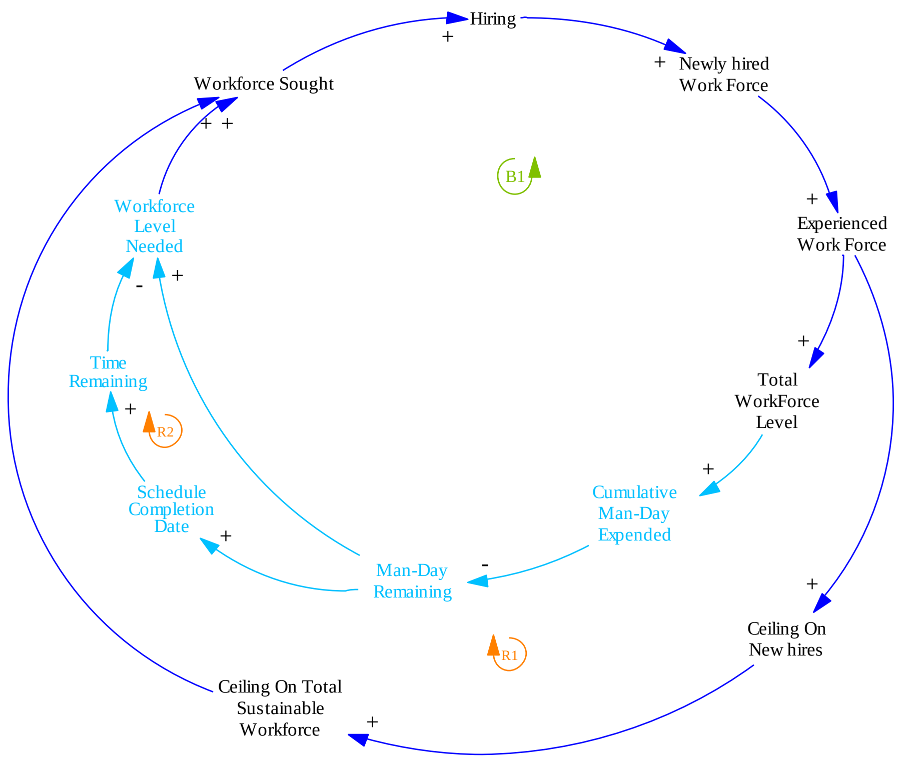


Figure 8. The causal loops of the human resource management subsystem

B1: The first balancing loop that is identified in this CLD considers hiring policies. Resource managers hire new people as the projects starts, to fill vacancies. When more people are hired, the amount of newly hired workforce increases. The newly hired workforce is trained and assimilated into the project, and this is highlighted by the positive relationship between newly hired workforce and the experienced workforce with a delay. The new hires become experienced personnel through experience and training. These two variables influence the total workforce level. When the total workforce increases, the Cumulative Man-Day expended (Manpower allocation) on the project increases as well. The relationship between the two variables is positive. When man-days have been expended the man-days remaining (control) decreases subsequently. The decrease in man-days remaining will indicate the need for a larger workforce through workforce level needed (planning), and the loop is closed as this indication provides adjustment to hiring.

R1: The first reinforcing loop identified focus on the training capacity and ceiling for new hires. The initialization of the project leads to hiring of workers that through learning and training becomes experienced workers. The process of training newly hired workforce includes the experienced workforce. The previous CLD identified this relationship. Training of new personnel is carried out by the experienced personnel. When more workers are classified as experienced personnel, more new hires can be trained and assimilated. This is expressed in the ceiling of new hires variable. More experienced personnel can train and assimilate more new workers. This increases the ceiling on new hires and the ceiling of the total sustainable workforce. This results in, an increase in the workforce level sought by management as they have the opportunity to add more people in order to finish the project on schedule. The loop is closed as the new workforce level sought provides new hiring strategies.

R2: The second reinforcing loop is linked to hiring and schedule considerations. When the project has been initialized and the workforce hired, the amount of man-days remaining (Control) decreases in tact with the increased workforce. This in turn influences the schedule completion date (Planning) and the time remaining on the project (Planning). From the time remaining, the workforce level needed (Planning) to finish the project on time is calculated. The loop is closed as the calculation leads to an adjustment in the hiring-policy by management.

4.2.4 The software production subsystem

The software production process consists of development (design and coding), quality assurance, rework and system testing process that are the major activities of a software development project. This subsystem is alone too complex to explain as one piece, therefore it is broken into four subsectors: Manpower Allocation, Software Development Productivity, Quality Assurance and Rework, and Systems Testing (Abdel-Hamid, 1991: p.69).

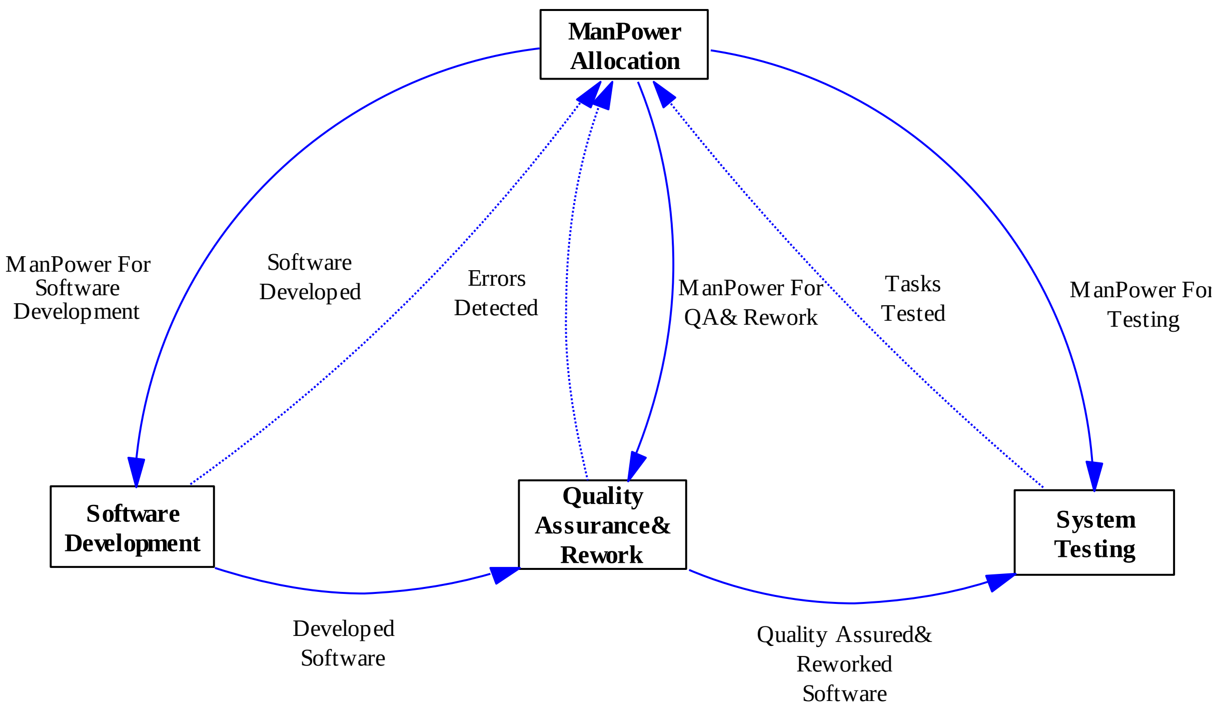


Figure 9. Software production subsystem

The figure above provides four central sectors in regards to software production. Manpower is allocated by management to the three different efforts running through the development process: Software Development, Quality Assurance and Rework, and Systems Testing. In the first step management plans ahead for the efforts needed, and allocate their resources accordingly.

Tasks are allocated to the development team which initiates the process by designing and coding the software. The developed software is then sent to the QA and Rework team. The process of quality assurance and rework runs in tandem with software production. The QA effort is an attempt to assure quality, were the production-team meet with managers and staff and work through the tasks completed. This is done through several techniques including walk-throughs, reviews, inspections, code reading and integration-testing. Through these steps, errors will be identified and sent to rework. Management will assign manpower to the rework-effort based on the total amount of errors found.

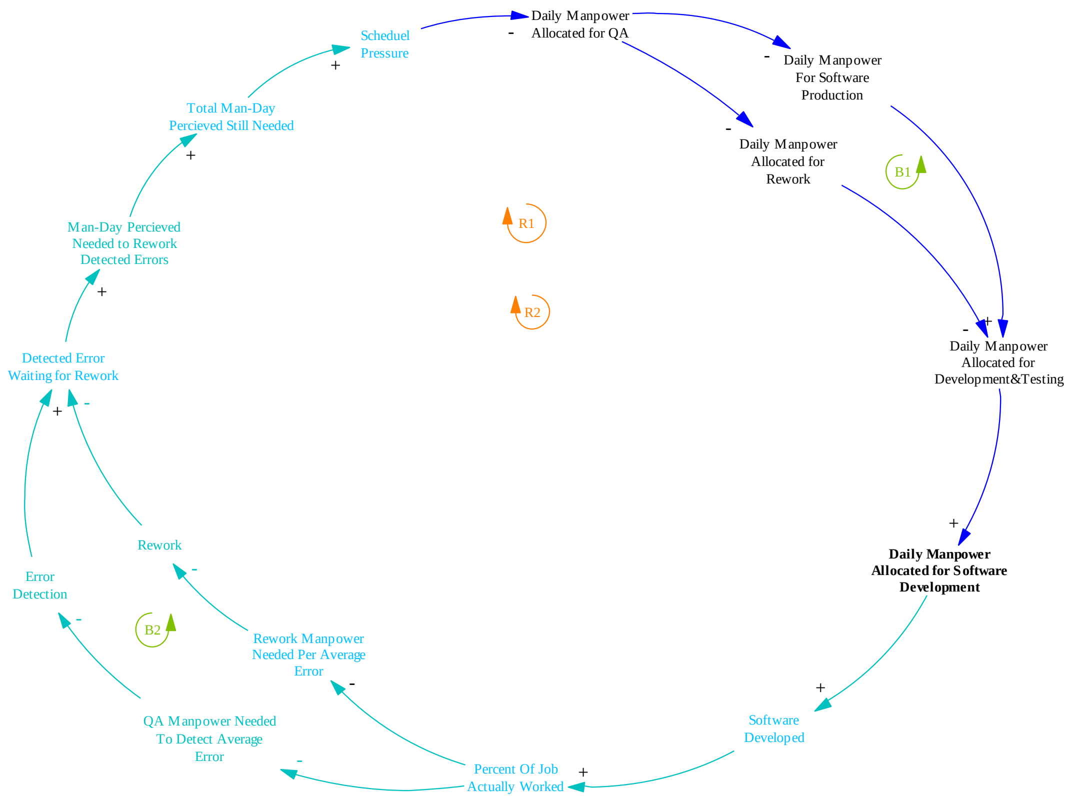
The quality assured and reworked software is sent to testing. When a project reaches a certain stage of development, manpower is allocated from software production to testing. The objective of systems testing is to verify that all components of the product mesh properly, and that the overall function and performances are achieved. The manpower transferred from development to testing is gradual during the last stages of development, but in the end the entire available workforce is allocated to testing. The objective of testing is then to test all tasks that have been developed.

So far this holistic model represents the holistic approach of Abdel-Hamid. From the figure depicted above, one may draw the conclusion that in regards to detail and explanatory power it is still limited. The actual variables and relationships have not yet been identified, and the representation is crude. The processes depicted above, provide manpower-allocation with feedbacks on their progress. Each sub-category’s effort provides status-reports, and manpower is allocated accordingly. So far, the holistic approach provides the main general ideas, but the nature of the relationship between variables is hidden. In order to identify them further and increase the understanding of the relationship, we need to open the subsectors and provide causal loops including the main variables, feedback-loops and processes.

4.2.4.1 The manpower allocation subsector

The manpower allocation subsector provides the information on how management allocates their different resources between four major efforts. Manpower management allocates resources between Quality Assurance (QA), Rework, Development and Testing. During the software production phase there are two main areas that management must allocate their resources between, the actual development of the software, which normally contains the main bulk of the total staff allocated, and the quality assurance and rework process.

The challenge for management is to allocate the correct ratio between these two sectors in order to utilize their manpower efficiently (Abdel-Hamid, 1991: pp. 69-75). Initially, manpower is allocated between software development and quality assurance and rework. As the program progresses, the testing effort is initialized. Staff from production is transferred to testing continuously until the final testing stage. In the end, the entire production staff is allocated to testing. The figure below depicts the Manpower Allocation Subsector:



Daily Manpower Allocated for QA
Daily Manpower For Software Production
Daily Manpower Allocated for Rework
Daily Manpower Allocated for Development&Testing
Daily Manpower Allocated for Software Development
Software Developed
Percent Of Job Actually Worked
Rework Manpower Needed Per Average Error
QA Manpower Needed To Detect Average Error
Rework
Error Detection
Detected Error Waiting for Rework
Man-Day Percieved Needed to Rework Detected Errors
Total Man-Day Percieved Still Needed
Scheduel Pressure
R1
R2
B1
B2

Figure 10. The causal loops of the manpower allocation subsector

B1: The first balancing loop identifies the central variables as manpower is allocated. It focuses on quality assurance manpower allocation, production manpower allocation, rework effort and process and schedule pressure.

The first allocation made by management, is to allocate manpower for quality assurance. When quality assurance manpower increases the production manpower decreases. When less manpower are available for the production process, less people are available for software development and testing. When less manpower is allocated to production, and development and testing less software is developed. (Development productivity)

Less software produced decreases the percentage of the job actually worked. (Control) This variable identifies the relationship between the design phase and coding phase during software development. The process of software development is evenly split between designs and coding. The first 50 % of production belongs to design, while the last 50% to coding.

When the percentage of work is less, the project is in the design phase. The rework manpower needed per average error increases. (Quality assurance and rework) With increased efforts needed to rework the average error, the error rework (Quality assurance and rework) decreases. Less rework increases the errors waiting for rework (quality assurance and control). When there are more detected errors waiting for rework, management’s estimate of man-days perceived needed to rework detected errors (Control) increases. This forecast is taken into consideration when the man-days perceived still needed on the project are calculated (Control) upwards. The increased need for man-days on rework creates increased schedule pressures (Control), and the loop is closed as manpower allocation to quality assurance decreases as a result of the increased in schedule pressure.

R1: The first reinforcing loop in the system concerns the relationship between quality assurance manpower allocations, rework manpower allocation, rework effort and process and schedule pressure. When more manpower is allocated to the quality assurance effort, less manpower is available for the rework effort. When less manpower is allocated to rework, more manpower is available for development and testing. The rest follows the same logic as for loop B1 effort.

More daily manpower for software development implies more software developed. More software developed, a higher percentage of job actually worked. Moving from design to coding phase, leads to a situation where the manpower that is needed to rework an average error decreases. Design errors are harder to detect, and also harder to rework. The average manpower needed to rework a design error is greater than the average manpower needed to rework a code error. This is indicated by the negative relationship between the percentage of job done variable and the manpower needed to rework average error variable.

The effort needed per error decreases, but the total amount of errors reworked increases (quality assurance and rework). This is also reflected in the negative relationship between average manpower needed to rework an error and the number of errors reworked. When errors are reworked, the numbers of errors are waiting for rework decrease. This provides a new estimate that is decrease in the man-days perceived still needed for rework, and subsequently decrease in perceived man-days still needed. The decrease in schedule pressure reacts to these estimates, and the loop closes as the allocated manpower for quality assurance increases in react to decrease in schedule pressure.

B2: The second balancing loop of this system is linked to the relationship between allocation manpower for quality assurance, allocation for manpower for rework, error detection effort and process and schedule pressure.

When management increases their allocation of manpower to quality assurance, the manpower available for rework decreases accordingly. The rework effort staff is allocated from the production side, and therefore there is a negative relationship between these variables. Less manpower available for rework provides more available staff to development and testing. More daily manpower allocated for software development increases the rate at which software is developed.

When software has been produced, and the percentage of job done increases, the quality assurance process is influenced by it. The average manpower needed to detect average error decrease, at the same time as the number of errors detected increase. This leads to an increase in errors waiting for rework, and the estimate for perceived man-day needed for rework reflects this increase. The increase in man-day needed for rework affect the perceived man-day still needed until completion, and the loop is closed as in crease in schedule pressures decrease the manpower allocated for quality assurance.

R2: The second reinforcing loop of this system follows the same logic of the loop B2. The link here however, is to the relationship between the manpower allocation for quality assurance, the manpower allocation for software production, error detection effort and process and schedule pressure.

When staff is allocated more to the quality assurance effort, less staff is available for software production, which implies less manpower allocated for development and testing. Less manpower allocated for software development, leads to a decrease in software developed. Less software developed provides less percentage of job actually worked, and the quality assurance manpower needed to detect the average error increases. The amount of errors detected decreases as a result and so thus the detected errors waiting for rework. Less errors waiting provide less man-days perceived needed to rework, and less total man-days perceived still needed. This reduces schedule pressures, and the loop closes as more manpower is allocated to the quality assurance effort.

4.2.4.2 The software development productivity subsector

During the software development phase, it is vital for management to conceive and predict the rate at which software is developed. The development rate hinges on several factors and is therefore more complex than just being a function of how much manpower is allocated.

The software productivity subsector focus on the overall productivity of the workforce on a project. There are several factors that influence the overall working efficiency connected to motivational factors, overtime and schedule pressures. In this subsector one differentiate between potential productivity and actual productivity. Actual productivity is expressed as the potential productivity subtracted losses due to faulty process. (Abdel-Hamid, 1991: pp. 77-94) Therefore one find a complex set of factors that must be considered in order to understand productivity. These factors include personal motivation, group motivation, schedule pressures and learning. In the figure below I present the CLD for the software development productivity subsystem:

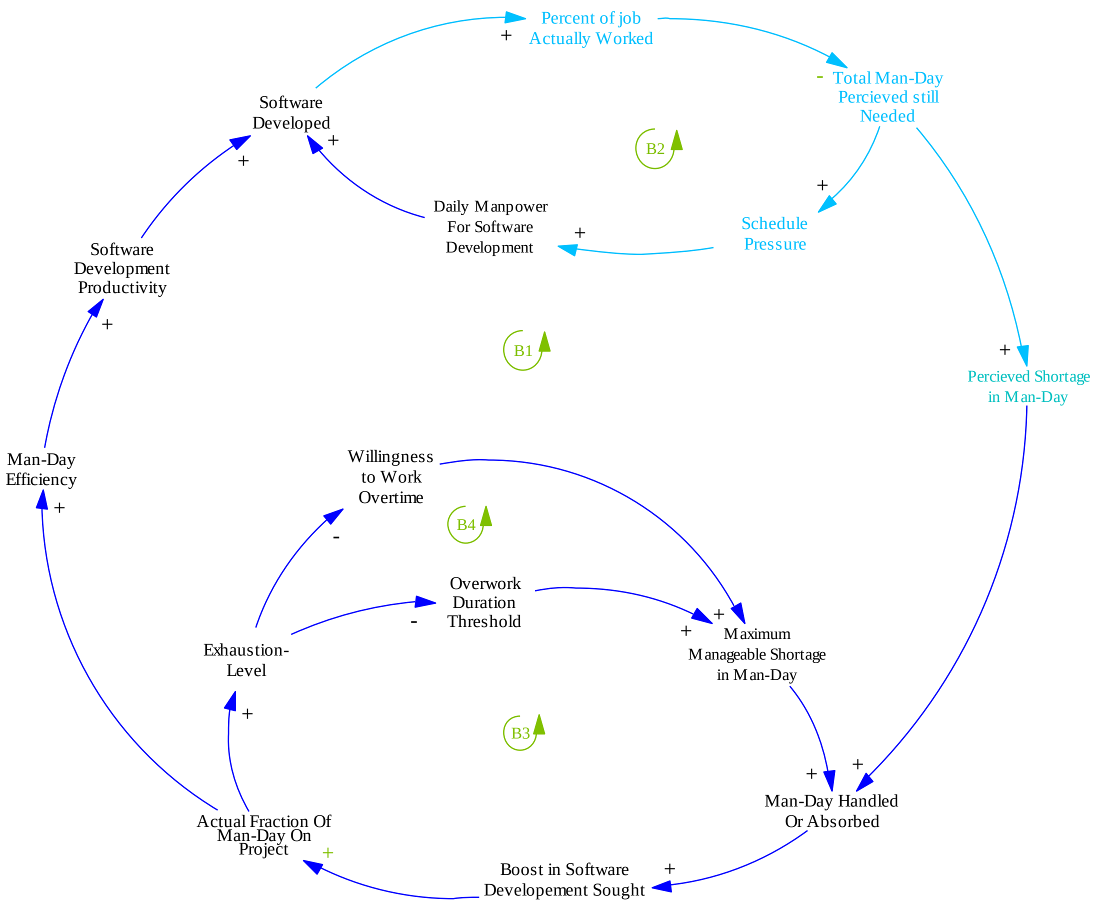

Percent of job Actually Worked
Total Man-Day Percieved still Needed
Software Developed
B2
Daily Manpower For Software Development
Schedule Pressure
Software Development Productivity
B1
Percieved Shortage in Man-Day
Man-Day Efficiency
Willingness to Work Overtime
B4
Overwork Duration Threshold
Exhaustion-Level
Maximum Manageable Shortage in Man-Day
B3
Man-Day Handled Or Absorbed
Actual Fraction Of Man-Day On Project
Boost in Software Developement Sought

Figure 11. The causal loops of the software development productivity subsector

B1: The first overall loop for this subcategory is the large external loop considering the amount of software produced, and the influences by the work remaining and then man-day still needed on the efficiency of production.

As software is developed, it has a positive impact on the percent of job actually done. When the software is being developed, the man-days perceived still needed (control) reduces in tandem with the percentage of the job actually done. In turn any perception of a shortage in man-day (control) according to schedule is reduced when software is developed according to the plan. Hence the Man-days handled or absorbed reacts in a positive relationship with the perception of a man-day shortage. If there is no shortage, there is no increase in man-days needed to handle or absorb the shortage. The rate at which management seeks to develop software per man-day hinges on the perception of a shortage of man-days. When there is no shortage of man-days or the project is ahead of schedule the sought level of software development remains the same or slightly adjusted downwards as the project is moving ahead. If there is a shortage however, the production per man-day needs to be increased in order to close the gap. Therefore the level sought will be boosted when a project is behind schedule.

The actual fraction of work spent on the project derives from the perception of how workers spend their time on the job. In an ideal situation, a group of staff will at any time work at peak efficiency. However, in a real work environment time is spent on different activities than just production.

The phenomenon of slack-time is important concerning overall productivity. Slack time is defined as the time lost to non-project activities. These activities include coffee-breaks, email reading, and personal activities. These are calculated as losses in man-hours. During the software development process, some hours will be lost to these activities and will reduce the actual fraction of time spent on the production of software.

When a project is behind schedule and management provides pressure by increasing the level of software development sough, the workers respond with cutting their slack time. Research shows that workers tend to cut their slack time, and devote more concentration to the work at hand when the pressure increases. Workers increase their man-hours in order to bring the project back on schedule. For a while this effect will increase the actual productivity as workers not only work harder during working hours, but also log overtime hours in order to close the time-gap.

Therefore there is a positive relationship between the two variables. When the level of sought software development decreases, slack-time increases. Hence the fraction of actual man-day spent on development decreases. When pressure increases the slack time is cut and people log more time on actual production activities. Logically this holds a positive relationship with man-day efficiency as less effort is put into each hour of work, then overall software development productivity decreases. The loop closes as the software developed rate decreases and the percent of job actually worked follows suit.

B2: The second balancing loop of this sub-system is the relationship between software developed, manpower allocation for software development and schedule pressures.

When percent of job increases due to more software developed, the total man-days perceived remaining decrease. When production leads to less man-days perceived still needed (control), the schedule pressures decreases as a result. Less schedule pressures leads to less manpower allocated for software development due to less effort needed. With less manpower for software development, the rate of software developed decreases. When software developed decreases, the percent of job actually worked decreases as a result and the loop is closed.

B3: The third balancing loop in this system concerns work pressure and the effects of increased working pressure on actual fraction of man-day on the project.

From the first loop we see that when there is a perceived shortage of man-days, the Man-Days absorbed or handled increases. This is done by the existing workforce as described under slack time. If a project is running late, the workforce log more hours, cut slack time and work overtime in order to bring the project back on track. Hence, increased man-days handled or absorbed leads to a boost in the software development sought. This leads to an increase in the fraction of man-day spent on the job per day. This brings a positive effect on productivity for a while, but there are some negative effects as well.

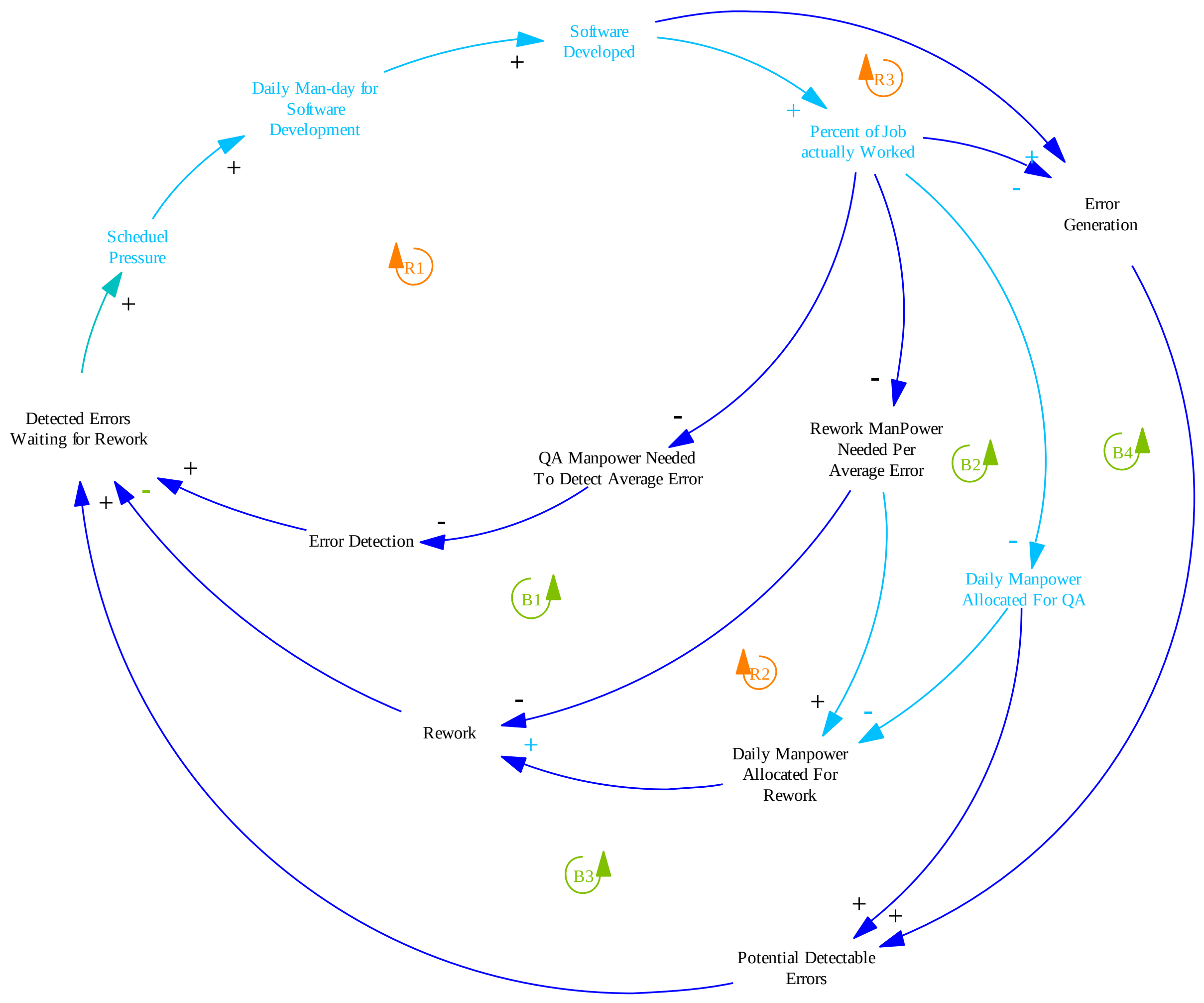
With increased work pressure, there are adverse effects to overall productivity, as the above normal situation causes strain to the workforce. There seem to be an overwork threshold, and an exhaustion factor limiting the boost in actual fraction of man-day on the project. According to the interviews conducted by Abdel-Hamid, staffs and managers are willing to increase their work-intensity within certain limits. If the need for overtime and increased labor-intensity is within these limits, there will be an overall boost from all staff in order to get the project back on track.

In the figure we see as a result that increases in work pressure leads to an increase in man-day handled or absorbed in the project. This leads to an increase in the fraction of actual man-day spent on the job that increases the exhaustion level. When staff feels exhaustion from overwork pressures, the overwork threshold is decreasing. Therefore there is a negative relationship between exhaustion level and the overwork duration threshold. This decreases the manageable shortage in man-day and man-days handled or absorbed and the loop is closed.

B4: The last loop of this system concerns willingness to work overtime and exhaustion level. When a project is behind schedule, and increase in man-day handled and absorbed creates work pressures, the exhaustion-rate increases. As explained above, workers and managers are willing to increase their work intensity within certain limits. If the need for overtime is within these limits, there will be an increase in productivity. However, there is a threshold for what workers and managers perceive is “within the limits”. Exhaustion over time will lower this limit and the willingness to work overtime decreases. With a decrease in willingness to work overtime, the maximum manageable shortage in man-days decreases as a result. The loop is closed as the man-day handled or absorbed decreases.

<u>4.2.4.3 The quality assurance and rework subsector</u>

The development of software systems involves a series of production activities where the possibility for interjection and human fallibilities are substantial. Errors may occur at the beginning of the production process where the objectives of the software system has not yet been clearly or erroneously specified as well as during the production process when set objectives are mechanized. It is vital that the finished product performs to the functional standards intended by the architects, with as little faults or errors as possible. To achieve this standard of quality, quality assurance is carried out in tandem with the software production phase. The CLD below depicts the QA subsystem:



Software Developed
Daily Man-day for Software Development
Scheduel Pressure
Detected Errors Waiting for Rework
Percent of Job actually Worked
Error Generation
QA Manpower Needed To Detect Average Error
Rework ManPower Needed Per Average Error
Error Detection
Daily Manpower Allocated For QA
Rework
Daily Manpower Allocated For Rework
Potential Detectable Errors
R1
R2
R3
B1
B2
B3
B4

Figure 12. The causal loops of the quality assurance and rework subsector

R1: The first reinforcing loop in this subsector concerns the software development progress and the quality assurance effort and process.

Increase in software developed provides an increase in the percent of job actually worked. Therefore as production moves from design to coding, the quality assurance manpower needed to detect an average error decreases. This follows the logic presented earlier under the manpower allocation subsector. When errors are easier to detect, the same level of quality assurance staffs can detect more errors and the error detection rate increases. When more errors are detected, the amount of errors waiting rework increases as well. When the amount of detected errors waiting for rework increases, it has an increasing effect on schedule pressures. With more schedule-pressure, the daily effort to do the tasks increases. This leads to an increase in the daily man-day needed for software development. The loop is closed as the daily man-day needed for software development increases the rate of software developed.

B1: The first balancing loop in this sub-sector concerns the software development progress and the rework effort and process. The logic behind the loop is similar to the description for the quality assurance effort and process in loop R1.

When software production increases, the percent of work actually done follows suit. This moves the project from design to coding, and less manpower needs to be allocated to rework the average error. The same amount of rework staff can rework more code-errors than design-errors due to the different error nature. The more errors that are reworked, the less errors are waiting to be reworked. Less detected errors waiting for rework has a positive impact on schedule pressure as the pressure decreases accordingly. The decreased schedule pressure leads to less daily effort to do the tasks, and subsequently less daily man-day for software development are needed. The loop closes as the less daily man-days needed for production leads to less software developed.

R2: The second reinforcing loop in this system is linked to the software development progress and the rework effort and process and the overall daily allocation of manpower to rework.

The software developed increases, and the percent of actual job worked increases. The project moves from design to coding, and coding errors are easier to rework. When the rework manpower needed per average error decreases, the management's daily allocation of manpower to the rework effort decreases as well. When the daily manpower allocated for rework decreases, so will the rework rate. The relationship is therefore positive between these two variables.

With less rework done, detected errors waiting for rework increase. This happens as production is still underway and new errors are detected by the quality assurance effort. More detected errors waiting for rework will increase the schedule pressure, leading to more need of daily man-day allocated to production. The loop is closed as the production rate of software increases from the increased daily man-days allocated.

B2: The second balancing loop concerns the daily allocation of manpower to the quality assurance effort and its effect on the daily allocation manpower to the rework process.

As in the previous loops increase in software developed leads to more percentage of job worked and shifting from design to coding phase of development. When the coding effort provides more passive errors that are easier to detect, the manpower allocated to the quality assurance effort decreases. Since the less amount of quality assurance can be assigned to find and report errors, the total manpower needed is perceived by management is smaller.

This in turn leads to more resources available for the rework process. Less daily manpower allocated to quality assurance gives more manpower to rework and provides more rework carried out. In turn this leads to less detected errors waiting for rework and subsequently less schedule pressure. Less schedule pressure causes the daily manpower for production to decrease, and the loop is closed when software developed decreases as well.

R3: The third reinforcing loop in this subsector looks at the relationship between software production and error generation. The increase in software developed, provides an increase in error generation. With more errors generated, the potential detectable errors increase as well. This increases the detected errors waiting for rework, and schedule pressures increase as well. With more schedule pressure, the need of daily effort to do the tasks increases. As a result daily man-day for software development increases, and the loop closes as software developed increases.

B3: The third balancing loop of the system describes the mechanism behind allocation manpower for quality assurance and the potential for error detection. As software developed increases, the percentage of job actually done follows suit. Moving from design to coding phase with an increased percentage in tasks done provides a decrease in the daily manpower allocated for quality assurance. Less manpower and effort for quality assurance translates into less potential detectable errors. Less potential detectable errors lead to less detected errors waiting for rework. Decreased errors waiting for rework provides less schedule pressure, and less need of effort to do the tasks. Thus, daily manpower for software development decreases as well. Less manpower for software development provides less software developed and the loop is closed. Here we see a potential risk for management as the lower quality assurance effort allows errors to remain undetected.

B4: The final balancing loop in this subsystem depicts the relationship between the percentage of job done and error generation. When software developed increases, the percentage of job is actually done follows. Moving from design to coding phase of development, leads to a decrease in error generation. The coding-tasks are in themselves easier to accomplish than design-tasks. Since coding-tasks follow the set design-blueprint they are more straightforward and errors are easier to avoid or detect by the programmer while the task is accomplished. Less errors generated, leads to less potential detectable errors. This in turn leads to less detected errors waiting for rework and the schedule pressures are decreased as a result. Less schedule pressures decrease the need of daily effort to do the tasks and so daily man-day for software development, and the loop is closed as the reduction in daily man-days provide a reduction in software developed.

4.2.4.4 The system testing subsector

System testing is the final subsector of the software production cycle. Errors that are not detected by the quality assurance efforts and escape or bad fixes due to faulty rework remain in the product until the testing phase. Undetected errors do not lie dormant until they are detected in the systems testing phase. This is highlighted in the model by constructing two separate processes for active errors and passive errors, where the growth and density of errors in the end influence the testing phase itself. Errors are divided into active and passive errors. Passive errors are dormant until detected in the testing phase, while active errors continue to create subsequent errors. Therefore the natural starting point is the undetected active errors from the quality assurance effort. The figure below depicts the testing subsector CLD:

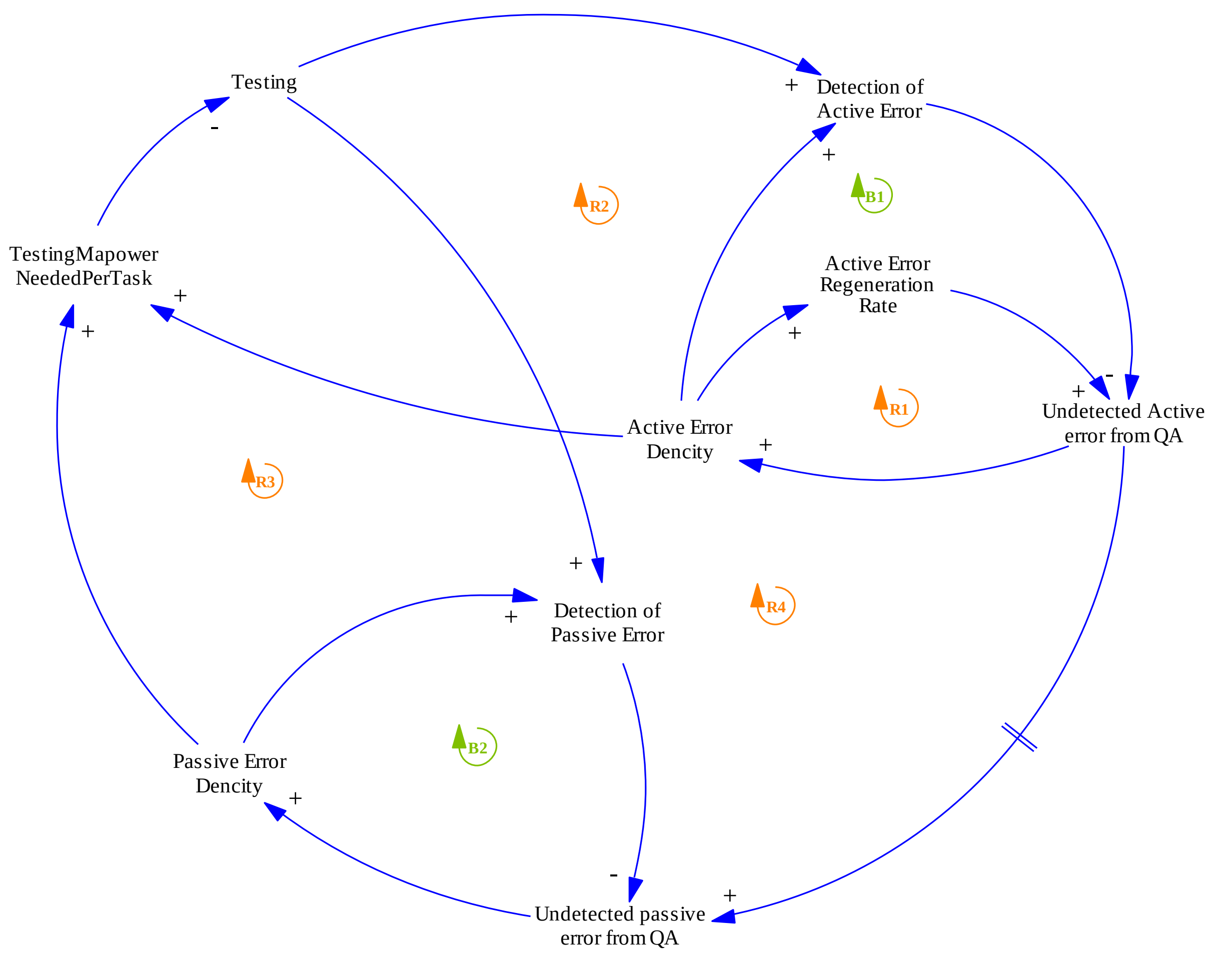

Testing
Detection of Active Error
TestingMapower NeededPerTask
Active Error Regeneration Rate
Active Error Dencity
Undetected Active error from QA
Detection of Passive Error
Passive Error Dencity
Undetected passive error from QA
R1
R2
R3
R4
B1
B2

Figure 13. The causal loops of the system testing subsector

R1: When undetected active errors from the quality assurance and rework effort increase, the active errors density increases as well. In the chapter on testing from Abdel-Hamid’s thesis we learned that the active error regeneration rate depicts that of bacterial growth. Increased active error density leads to an increased active error regeneration rate, which in turn leads to an increase in undetected errors from quality assurance and rework process closing the loop.

B1: The first balancing loop of the CLD concerns the detection of active errors. When errors undetected by the quality assurance and rework effort increases, the density of errors increases accordingly. When more errors are apparent, the likelihood of their detection increases as well. In turn, the increase in detected errors reduces the amount of undetected errors from the quality assurance and rework effort, and the loop is closed.

R2: When the amount of undetected errors from quality assurance and rework process increases, the density of errors increases as well. The increase in error density provides more errors per task, and more testing manpower is needed to test an average task. More testing manpower needed per task, provides a lower testing rate. Less testing provides less detection of errors, and more undetected errors pile up in the undetected errors from the quality assurance and rework effort as a result and the loop is closed.

B2: The second balancing loop concerns the detection of passive errors, and the logic follows the one behind active errors. Active errors will keep multiplying themselves for a certain amount of time. Active errors retire after one or two "generations" of errors and become passive errors. An increase in the number of undetected passive errors, leads to an increase in the passive error density. More passive errors create more detection of passive errors. The loop is closed when the amount of passive errors detected decrease the number of undetected passive errors from quality assurance and rework process.

R3: The third reinforcing loop concerns the passive error density and testing process. When undetected passive errors are fed into the system from the quality assurance and rework process and active errors retire the density of passive errors increase as well. This in turn leads to an increase in manpower needed to test a task, as every task will hold a higher density of passive errors. This in turn slows down the testing process, and fewer errors are detected as a result. The loops is closed as passive errors from quality assurance and rework process increase as a result of higher passive error density and less testing performed.

R4: The fourth and final reinforcing loop concerns the detection of both passive and active errors in the subsector as a whole. When more active errors are sent undetected to the testing phase, the rate of undetected errors increase. After some delay the active errors are retired and become undetected passive errors instead. The density of undetected passive errors increases as a result, and the manpower needed to test an average task is higher. More manpower spent per average task leads to less testing, and more active errors slip through. The loop is closed as more undetected active errors slip by and end up in the pile of undetected errors from the quality assurance and testing effort.

4.2.5 The Planning Subsystem

The planning phase is the third subsystem from Abdel-Hamid's holistic model. The planning phase is the starting-point for any software project. Before production and control is carried out, the fundamental framework on the project's size, budget frame and scope is chiseled out. The estimated completion date (measured as man-days required) and workforce density is established.

This is a key subsector for the managerial effort on the software project as a whole. Policies regarding working hours, work force density and completion date are considered and set here. These policies are then fed back to all the other subcategories in the model. Therefore planning-adjustments hinges on several different factors addressed in all the previous sectors. For example, if there is a difference between the indicated workforce needed for the project and the actual workforce, people may be hired or fired in order for the gap to be closed. The CLD for the subsystem is presented below:

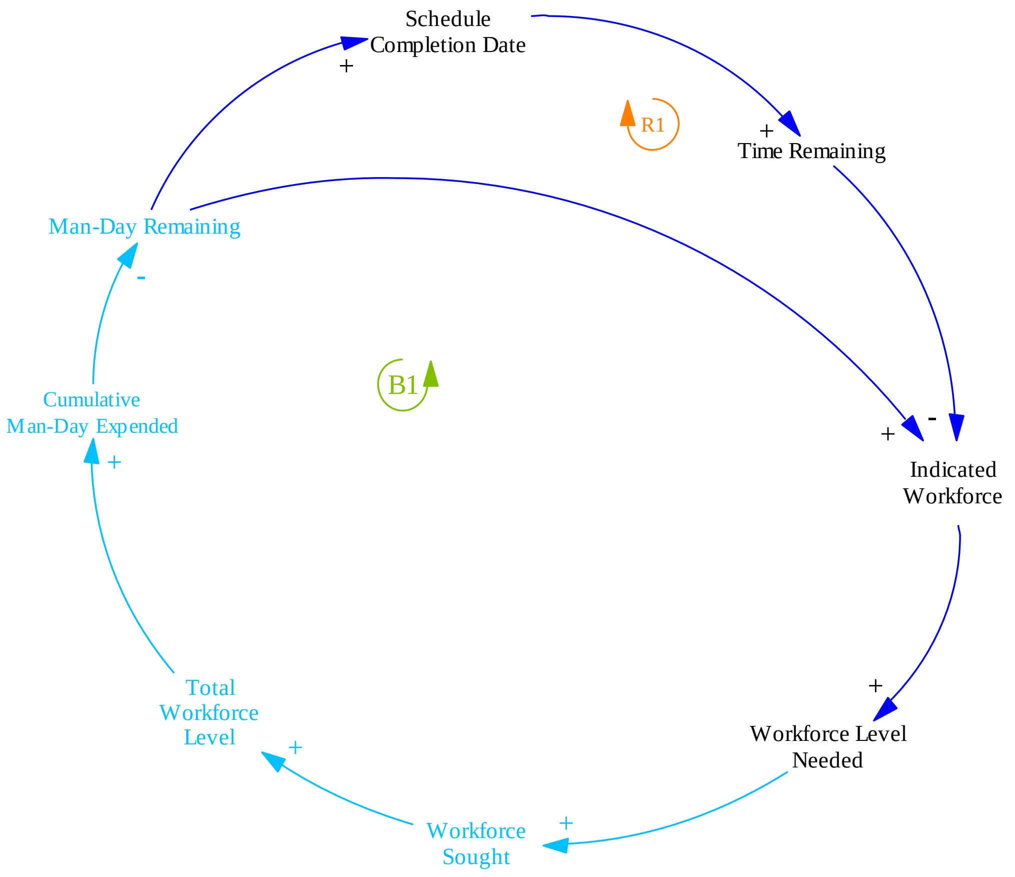


Figure 14. The causal loops of the planning subsystem

R1: The first step in the planning phase is to formulate a schedule for the project's completion. Therefore it's a natural starting point for tracing the activity through the first loop. When the scheduled completion date is increased, the time remaining increases as well. With an increase in time remaining, the indicated workforce needed to finish on time decreases. This leads to a subsequent decrease in the work force level needed, and the work force sought by management decreases (human resource management). The decreased level of workforce sought provides a similar decrease in the total workforce level (human resource management). When management seeks fewer employees they hire less. Decrease in the total workforce level provide a decrease in the total cumulative man-days expended (manpower allocation), and the man days remaining in the project increases as a result. The Man days remaining (control) gives an indication on the scheduled completion date and the loop is closed as the increase in man days remaining provides an increase in schedule completion date.

B1: The balancing loop in this CLD is concerning the man-days remaining and the indication for the workforce needed. With an increase in the indicated workforce, the workforce level needed increase as well. This leads to an increase in the workforce level sought, and the total workforce level. Increases in the total workforce level give a decrease in the cumulative man-days remaining. The decrease in man-days remaining closes the loop as it leads to a decrease in indicated workforce.

4.2.6 The controlling subsystem

The control function in a software development project comprises of three elements. First element is measurement, detection of what is happening in the activity that is currently being controlled. This leads to the second element, evaluation. This entails an assessment of the significance of the activity being controlled. This is usually done by comparing information on how the activity is progressing with a preset standard or expectation on how the activity should be progressing. The last element is then communication. To report the measures and assessments so that behavior can be altered if the need is apparent.

In a software project, progress is measured by the number of resources consumed, or tasks completed, or indeed both. Measurement calculates the number of man-days still needed to complete the project. These include production time, time for developing quality assurance tasks, to rework errors and test the software. This measurement is then compared with the actual man-days remaining before the deadline. If there are more man-days remaining in the calculation than actual man-days available the project is behind schedule. This assessment in turn leads to management communication, where one solution can be to motivate the staff to work harder in order to get the project back on schedule. Other changes may be to extend the schedule itself (Abdel-Hamid, 1991: pp.117-119).

It is important to note in this subsystem the concept of tasks discovered, and how they fuel the adjustments in allocation, man-days perceived still needed and the production effort. Measurements of progress are a challenge for software management, and the dynamic relationship between underestimation of job-size and software production is identified here. The CLD in next page depicts the controlling subsector:

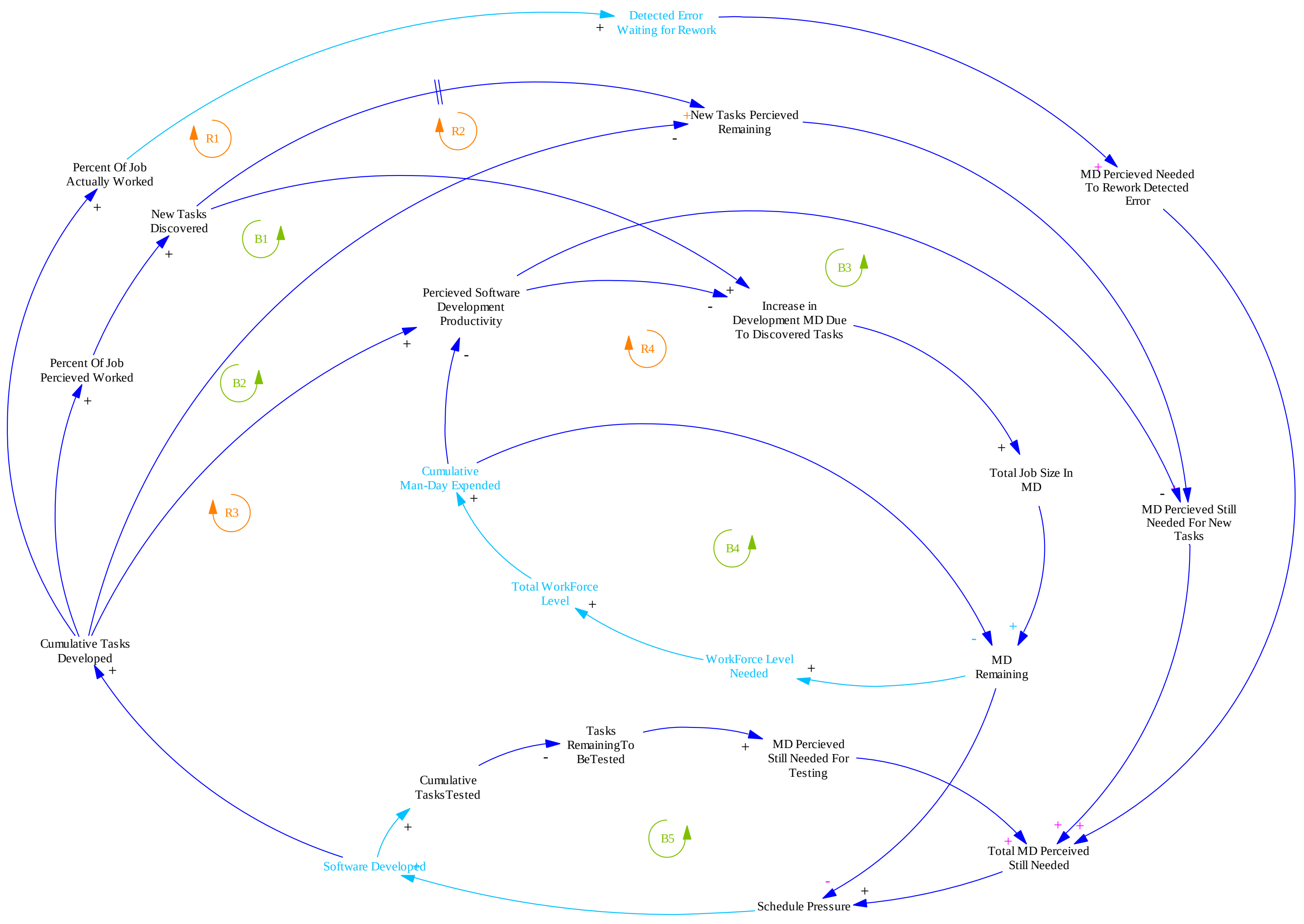

Detected Error Waiting for Rework
New Tasks Percieved Remaining
MD Percieved Needed To Rework Detected Error
Percent Of Job Actually Worked
New Tasks Discovered
Percieved Software Development Productivity
Increase in Development MD Due To Discovered Tasks
Percent Of Job Percieved Worked
Total Job Size In MD
MD Percieved Still Needed For New Tasks
Cumulative Man-Day Expended
Total WorkForce Level
Cumulative Tasks Developed
WorkForce Level Needed
MD Remaining
Tasks RemainingTo BeTested
MD Percieved Still Needed For Testing
Cumulative TasksTested
Total MD Perceived Still Needed
Software Developed
Schedule Pressure
R1
R2
R3
R4
B1
B2
B3
B4
B5

Figure 15. The causal loops of the controlling subsystem

R1: The first reinforcing loop of this subsector is focused on the cumulative tasks developed , error detection, man-days perceived needed for rework, schedule pressures.

As more tasks are developed and cumulated, quality assurance starts its effort. More cumulative tasks completed translates to more percent of the job actually worked, pushing the project from designing towards coding. In the coding stage the errors are easier to detect, so the increase in job actually worked increase the detected errors waiting for rework. With an increase in the detected errors, the man-days perceived needed to rework detected errors increase as well. When the man-days perceived needed to rework is increasing, the total man-days perceived still needed logically increases. When the man-days perceived still needed increases, the schedule pressure increases as a result.

When the schedule pressure increases, workers cut their slack-time and devote more time to the actual production of software. Hence, the schedule pressure increase leads to a boost in software developed. When more software has been developed the tasks are completed and compiled. The loop is thereby closed as the cumulative tasks developed increases in tandem with software developed.

R2: The second reinforcing loop in this system identifies the relationship between tasks developed and new tasks discovered. This is an important feature as it identifies the problem of underestimation.

In the base-model we see how an increase in the percent of job perceived worked move the project from design to coding in the same manner as percentage of job actually worked. In software management, there is a big issue of underestimation. In the earliest stages, management has few milestones to measure progress by. This leads to a situation where the perceived job size and man-days needed mimics the pre-planned schedules. When the project moves from design to coding, management and staff realize the true efficiency of the work so far. Secondly, they realize the true size of the project or the real amount of tasks needed (Abdel-Hamid, 1991: p.119).

This is illustrated in this loop in the following manner: When the cumulative tasks developed increases, the percent of job perceived worked increases. Increase in percent of job perceived worked leads to an increase in tasks discovered as the true size and tasks are revealed to management. When tasks discovered increases, the new tasks perceived remaining increases as a result with a delay. When the perception of tasks remaining increases, management increase their perception of man-days perceived still needed for new tasks. More man-days for new tasks translates to more total man-days perceived still needed, and schedule pressure increases as a result. When schedule pressure is high, the same effect as for loop R1 allows more software to be developed as a response of urgency. The increase in software developed increase the cumulative tasks developed as well. This closes the reinforcing loop R2.

B1: The first balancing loop of the system identifies the relationship between new tasks discovered, increase in development man-days due to discovered tasks, and total job size in man-day.

When more cumulative tasks developed, and the percent of job perceived worked increases subsequently, more tasks are being discovered. It leads to an increase in the development man-days due to discovered tasks. The schedule is lengthened by management to accommodate the increase in tasks. Therefore, the increase in total job size in man-days leads to more man-days remaining and subsequently to less schedule pressure. When schedule pressures are less, the software developed rate decreases as a result from the eased tension. Less software developed leads to less cumulative tasks developed and the balancing loop is closed.

B2: The second balancing loop of the system is looking at the relationship between cumulative tasks developed, new tasks perceived remaining, total man-day perceived still needed and schedule pressure.

When the cumulative tasks developed increases, management’s perception of new tasks perceived remaining drops as a logical consequence. With less new tasks perceived remaining, the man-days perceived still needed for new tasks decreases as a result. With less man-days perceived needed for new tasks, less overall man-days are perceived needed for the project as a whole. The drop in perceived man-days still needed provides less work-tension, and the schedule pressure decreases accordingly. Less schedule pressures leads to less software developed, and the cumulative tasks developed decreases subsequently. This closes the second balancing loop.

B3: The third balancing loop identifies a second important feature for controlling, namely perceived software development productivity. Development productivity connected to man-days perceived still needed, schedule pressure and cumulative tasks developed.

When the number of cumulative tasks developed increases, the perceived software development productivity increases. The reason for this perception is linked to the relationship between cumulative tasks and productivity. When a software project moves from design to coding, the number of tasks accomplished increases more rapidly. Workers learn more about the software they are creating, and the reduced effort needed per coding-task compared to design-tasks allows them to produce finished code faster. When more tasks accumulate faster, management perceives this as an increase in the overall productivity (Abdel-Hamid, 1991: pp.117-120). Management translates the increase in cumulative tasks to increase in productivity, and the man-days perceived still needed for new tasks decrease accordingly. When the perceived man-days for new tasks decrease, the overall man-days perceived still needed follow suit logically. Less man-days perceived still needed provides less schedule pressures, and subsequently the software developed decrease as well due to less tensions. Less software developed provides less cumulative tasks developed, and the balancing loop is closed.

R3: The third reinforcing loop of this subsector is linked to cumulative tasks developed, perceived software development productivity, total job size and schedule pressure.

When the cumulative tasks developed increase, we saw in loop B3 that management translates this into a perceived increase in software development productivity. When the perceived software development productivity increases, it leads to a decrease in development man-days needed due to discovered tasks. This is translated through both cumulative tasks developed and perceived software productivity.

Less development man-days needed due to discovered tasks leads to a decrease in total job-size calculated in man-days. Less total job size leads to less man-days remaining on the project and schedule pressures increases as the project is moving closer towards the deadline. More schedule pressures increase the software developed rate as workers are scrambling to complete the project on time. This leads to an increase in cumulative tasks developed and the reinforcing loop is closed.

R4: The fourth reinforcing loop links perceived software productivity to total man-days needed for discovered tasks, total job size in man-day and total workforce level.

When the perceived software development productivity increases, the development man-days due to discovered tasks decreases as a result. With less man-day needed due to discovered tasks, the total job-size in man-days decreases logically. Less man-days in total job-size translates to less man-days remaining. When less man-day remain, the calculated workforce level needed decreases as the project is moving towards completion. Less workforce level needed provide less total workforce level as management responds to the new calculated needed workforce. Less total work-force provide less cumulative man-days expended and perceived software development productivity increases closing the reinforcing loop.

B4: The fourth balancing loop of this system links the cumulative man-day expanded with man-day remaining in the project. Increased cumulative man-days expanded leads to less man-days remaining. Less man-days remaining provides a recalculation of workforce level needed by management, and it is adjusted downward accordingly. Less workforce level needed leads to less total workforce level and the loop is closed as a lower workforce level leads to less cumulative man-days expended.

B5: The last and final balancing loop in this CLD connects software developed with the man-days perceived still needed for testing, and schedule pressure. When there is an increase in the software developed, the cumulative tasks tested increases as well, as finished tasks are sent to testing. With more tasks tested, the tasks remaining to be tested decreases logically. Less tasks remaining to be tested decrease the man-days perceived still needed for testing which decreases the total man-days perceived still needed in the project. Less man-day still perceived needed decreases the schedule pressure, and the loop is closed as less software is developed as a result of more slack-time at low schedule pressures.

4.3 Summary

From this presentation of Causal Loop Diagrams (CLDs), we see how interconnected the different variables in a software development project are. In the introduction to Abdel-Hamid's doctoral thesis he points out that management have a big challenge due to this complexity (Abdel-Hamid, 1991: pp.4-6). In addition to this complexity, there are feed-back processes that will create different strengths between the loops, as a software project moves between the development stages. This is a vital contribution from the different feed-back loops I have identified. In the previous work, we could trace the variables across systems through the simulations and observe their behavior. With my CLDs, we can additionally see what processes are creating this behavior and how it changes over time.

In all the sub-systems and subsectors presented here, variables from other systems have been influential as well. Different loops create reinforcing effects and balancing effects and the sum of their strength explains the observed behavior. In order for management to understand and control the different effects, it is vital to understand how loop-strength change over time, and what effects the additional loops gain when management adjust their policies towards one specific area. In a software project all the different variables are interconnected, and a change in one policy will have an effect on all aspects of production simultaneously or with a delay. The CLD-tool together with the stock and flow diagram will now provide a model that is more tuned to these changes and consequently more able to identify them.

Chapter 5: Research Questions and Reference Mode

5.0 Introduction

In this chapter I will present my research questions and reference mode. I would like to stress that the main effort in my work will focus on the quality, rework and testing efforts. However, rather than isolating these segments of the model and develop them further, I wish to investigate how my enhancements or policy-changes influence the overall model and some of the key assumptions. Since software production is a complex system and different activities are interrelated, it would be more valuable to integrate my modifications into the whole model rather than narrowing my scope to a specific compartment. Hence, the two research questions that I have formulated and seek to answer through my research reflect this approach. My questions are related to the core assumptions of the original model, to the original conclusions that were drawn, and to policy conclusions concerning key factors such as manpower-distribution.

The reference mode presented in this chapter provides key graphs that depict the behavior of the base-model. In System Dynamics we seek to provide a dynamic characteristic for the problem under investigation. Reference modes consist of graphs and descriptive data that depict the behavior pattern of the problem. It shows how the problem arose, and how the problem might evolve over time. Reference modes are an important tool for providing an understanding of a problem for a long-term period rather than short-term observations (Sterman, 2003: p.90). My reference mode is presented as graphs showing the behavior of key stock and variable in the model related to defect removal.

5.1 Research Questions

Under this headline I will present the two research questions I seek to answer. These questions are directly linked to the base-model of Abdel-Hamid, and will act as guideposts for my reply. My first question focuses on the different subsystems of the base-model, and central assumptions related to the software development process within the model as a whole. My first question is:

<u>Q1</u>: *Are there any unrealistic assumptions in Tarek Abdel-Hamid's "Software Development Project management" model? If so, how we can modify those assumptions to make the model more realistic?*

Here I wish to draw out the quality assurance, rework and testing phases as areas of concern. There are four reasons for this choice. The first reason lies in the room for improvement highlighted by the author himself, and the function of quality assurance, rework and testing presented in the model. In the end of Abdel-Hamid's thesis he provides several areas that will require improvements in future research. One of the areas that will need enhancements is software quality (Abdel-Hamid, 1991: p.227).

My motivation for doing this research, rest on the possibility to achieve increased software quality. Quality of software products hinges on the three efforts of quality assurance, rework and testing, and will therefore be suitable for further enhancements. Further-more, in previous studies the analysis of the quality assurance efforts were largely focused as a cost-function of production. The question of allocation was only addressed according to cost-estimates and not by the nature of the process. Hence, important questions like motivational factors on the rework effort were never addressed. Therefore I find enhancement potential in the realm of quality assurance, rework and testing.

The second reason for choosing these three efforts rests with their pivotal position in regards to project costs and errors. From the literature review, we learn that quality assurance efforts are considerable in order to detect errors during the production process. These errors are then reworked at a later stage by the production teams. These quality check and rework efforts are very important in understanding costs and benefits regarding quality of software products.

Managements allocate resources to quality assurance, rework and testing. The manpower allocated to these efforts decrease the manpower available for production activities. It is very important to balance the allocation of manpower between the different efforts depending on the production process. With my enhancements, the need for re-allocation of manpower to testing at an earlier stage is a key consideration. Increased realism and new information available for management, new considerations on manpower allocation are necessary.

The third reason rests on the apparent lack of a feedback connection between the testing phase and rework phase in the model. In the base model systems testing is the final effort before the project is complete. However, from the literature we see that medium sized software projects run testing in tandem with rework and quality assurance at set stages. Therefore there is a major assumption that the testing-phase commences as an end process to production. As a result, the errors found at testing are not fed back into the production process to get reworked and corrected. This constitutes a dead-end in the model. This is a vital point for improving our understanding of the relationship between rework and testing that is currently missing in the base model. By addressing the relationship with rework and testing I will be able to increase the realism of the model, and it provides a valuable point of entry for policy-decisions.

The fourth and final reason rests on testing process. When we look at the literature on software production we find ample evidence for several stages of testing during a project's lifecycle. Looking at data provided from the U.S on testing in software projects, we find that only 2% of projects utilize only one testing stage. The most common number of testing-stages is 6, appearing in 21% of software projects (Jones & Bonsignour, 2012: p.324).

It also addresses a gap of knowledge in the literature as well. The importance of testing and the universal utilization of testing have been established in the literature as an important tool. But the focus on testing has not however, produced quantitative data on testing. Jones and Bonsignour note that "the lack of quantitative data does not speak well for the professionalism of the software industry. For more than 50 years finding and fixing bugs, have been the most expensive identifiable software cost driver. High defect levels are the primary reason for software schedule and cost overruns. Knowing the impact of defects on software projects and the impact of test cases and test effort on project performance is critical information." They argue that the lack of such data goes a far way of explaining the endemic situation with cancellations and schedule delays (Jones & Bonsignour, 2012).

The second question looks at the overall model behaviour and how my enhancements can provide additional improvements to its behaviour. The second question is:

Q2: *In terms of policy conclusion (for example: distribution used to resources), are there additional policies for instance, supplementary allocation of resources to quality assurance, rework and testing efforts that would improve the model behaviour?*

Regarding the question of policy inclusions, my enhancements will provide a basis for applying policy in the thesis. The first enhancement provides an expanded testing regime where software tests will be conducted during production. In the original model testing is only carried out as a final systems test. The literature on testing indicates that modern software projects carry out wide variety of technical tests during production in order to ensure product quality. Tests are conducted at certain milestones, and provide error-detection (Jones & Bonsignour, 2012). With an enhancement to the testing-effort, I will be able to calculate and provide additional policies regarding allocation of resources to the quality assurance, rework and testing effort. These policies should enable better software quality and still be cost-efficient methods. It is important that we achieve better software quality without creating severe additional costs for the project and the client.

The second enhancement to the model is a client-review system. In modern software projects the client is an important actor. With the increased use of customer acceptance testing, I provide a testing-regime that includes the client. The client-review will be split between a team allocated by the testing staff and personnel allocated by the client. The testing staff will present a piece of software or application to the client staff. The client's staff will carry out the review of the applications and provide their experiences with the software. These reviews are valuable because they provide an additional instance for error detection.

The empirical data on customer acceptance testing reveal an average defect removal efficiency of 30% (Jones & Bonsignour, 2012: p.320). It is important to note that the inclusion of the client in this regime is in accordance with both sides of the argument regarding the client. The client is allowed to participate in end-user testing, but will not provide technical fixes or solutions. The overall technical aspect of the software is kept exclusively in the realm of software development. The new client-review and the enhanced test regime will change the allocation of manpower for testing.

In the base-model, all members of the production staff are allocated to testing near to the end of the development cycle. With these new structures, the allocation for testing process will be allocated during the development phase and will be separated into different stages of testing, and between systems testing and client-review.

So far, the enhancements I have suggested have been linked to the testing effort. Testing is regarded as the most important tool for defect removal. For some projects, the testing effort is the only measure for defect removal. However, testing alone is an expensive and limited tool. Therefore, my third enhancement is linked to the incorporation of the quality assurance effort into the overall defect removal effort. Quality Assurance and Testing will now be linked with each other as tandem defect removal-tools. It is equally effective for defect removal to have pre-test efforts that identify errors and prevent further occurrences. There are several different techniques available. I have chosen a relative new approach called the capture-recapture method. The general idea stems from biology (Jones & Bonsignour, 2012). When errors are found in the quality assurance effort, they are first reported and sent to rework. Thereafter, the fixed pieces of code previously generating the reworked errors are tagged. After tagging, they are released back into the system and sent to testing. During systems testing after the classification stage, errors are found and identified by the testing effort. Deviating tasks that are found to contain tagged errors will be send back to rework, as they are a result of bad fixes. Deviating tasks that holds no tagged errors are sent back to the quality assurance effort for classification. Tasks that contain untagged errors are a result of escaped errors from the previous detection process.

During system testing after classification phase and during identification phase deviated tasks with tagged errors in, will be send back to rework as they contain the bad fixes errors but deviated tasks with no tagged errors in, will be send back to quality assurance process to get the error detection there first as they contain the escaped errors from previous detection process.

5.2 Reference mode

System dynamics models are built in order to characterize a problem, not a system. It is therefore a necessity to first identify the problem, the key characteristics of the problem and how the model can reproduce the observed problematic behavior. Thus, I will provide key stocks and variables as the reference modes, and show their behavior by graphs.

The focus in this research is the software crisis. Through the literature, we learn that software projects are hallmarked by late deliveries, cost overruns and poor quality. In Abdel-Hamid's original work, one sought to identify the general problem through cost functions and management decisions. The cost side of the problem was over-weighted. One of the main goals for any software project is to deliver high quality end products. However, recent research on this field reveals a continuous trend of poor quality software.

In my research the question of software quality will be paramount in order to set the quality issue first. From the recent literature on this field we see that lack of quality of process and quality of end-product are the roots to the software crisis rather than a symptom of the disease. Cost-overruns, delays and scrapped software are caused by poor quality. Yet, in the original work of Abdel-Hamid the concept of quality was not addressed as an underlying source to the observed problem. The change of focus in my thesis is reflected in the reference mode as the key stock and variable are related to defect-removal

The software crisis is costly for the industry and the end user. Examples from the Standish report, the ESSU report and the DNV report all portray the same picture. Every year several billions of dollars are wasted on software projects that are either running late, never finished or scrapped a short time after completion. In addition, several billions are spent every year to rework or maintain existing software that never performed like expected (ESSU: 2007, DNV: 2012, CHAOS: 2005). This is the monetary side of the problem. However, when software fails it causes other problems in society as well. Software is rapidly expanding into a product that runs almost every technical system in society. Low quality software in aviation, hospitals or other life-essential services could cause harmful and deadly situations. This has already been observed, by for example Jones and Bonsignour. They argue that medical mistakes in a hospital for instance are often a result of software failure (Jones and Bonsignour, 2012).

In the original model of Abdel-Hamid, the issue of software quality was to some degree omitted. One of the major assumptions of the model rests on the premise that errors undetected by quality assurance will be picked up by the major testing effort at the end. However, from the reported cases it is apparent that this is not the reality. The failed software projects under investigation are all branded with high amounts of errors. The Altinn-case works as an example, with a significant amount of errors embedded in the end product (DNV: 2012).

The error-factor is omitted in the base model due to two factors. The first factor is the focus on quality assurance and rework as an economic function, the second factor is due to the model boundary. The model keeps the end-product outside the model boundary. The additional costs of failing software or consequences of poor quality are outside the model scope. This provides a model that is only focusing on the software production aspect, and not the entire software cycle as a whole. This is a result of the limited information available at the time, and the over-weighted focus on economy in the industry itself. However, in retrospect it is apparent that the impact of poor software quality is costing industry and clients even worse substantially in long term. This factor must be calculated when we seek to improve software quality.

Software quality holds a second effect on the concept of the software crisis, namely the quality of process. The literature on quality states that poor end-products stems from poor processes during production. When an end-product holds errors or flaws it is a result of bad craftsmanship. When we seek to produce high-end products the entire process must focus on this goal. In the software industry we see evidence of lacking routines and process quality. This is again illustrated by the Altinn-example. Poor procedures lead to poor quality and errors. This is a very important factor for understanding the other symptoms of the software crisis as well. Unfortunately, this aspect is not in focus in Abdel-Hamid's model. Errors and continuous rework leads to more delays and schedule pressures. The continuous need to re-allocate manpower for different efforts in order to bring the project back on track all shows the sign of poor quality of process. Hence, important and central effects on the reference-mode's behaviors are the efficiency of different efforts during the production lifecycle and allocation of manpower between these efforts. They are very important in order to provide software quality and avoid software failures. The stock and variable that I will focus on are the "Detected Errors Waiting for Rework", and the "Detected Errors Rework Rate" through the project stages.

In order to construct a reference mode, I have to base the key features on the original model. There are few longitudinal studies on software quality, as the industry is relatively young and major changes have taken place in recent times. Abdel-Hamid's model was empirically tested during the DE-A NASA project. The model was able to reproduce the actual behavior of several key factors, such as manpower allocation during the production cycle, project schedule completion date and total cost of the project in man-day. It was also controlled towards an IBM project, and did reproduce the major overall behavior. Unfortunately, the original data series was not part of his report. Consequently, my reference modes are built on graphs from base simulation runs. These graphs are of course less accurate than raw data, but nevertheless provide a clear picture of the observed behavior. It is vital that my enhanced model reproduces this behavior, before solutions can be implemented.

The graphs depicted below represent the stock and the variable that are important factors for understanding the software crisis of the quality. They are connected to quality assurance of the product and detection of errors and rework of detected errors. These graphs illustrate appropriately the problems associated with misperceptions.

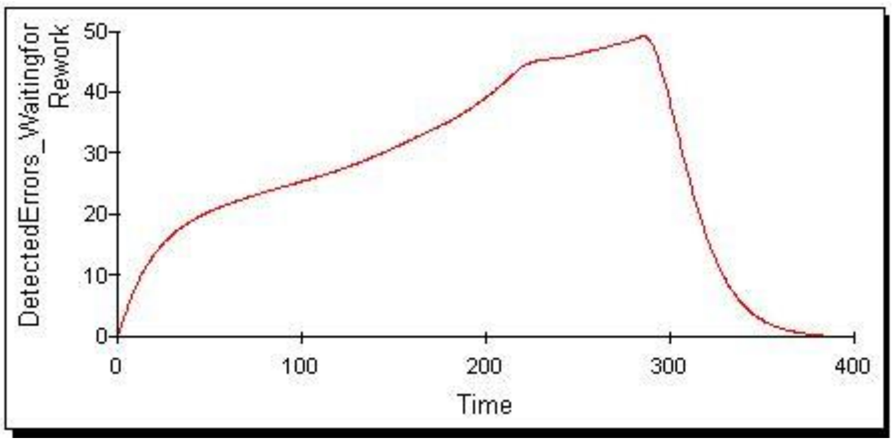


Figure 16. Detected errors waiting for rework

The first graph to be depicted in my reference mode is the "Detected Errors Waiting for Rework" shown in figure 16. The amounts of errors are detected and waiting to be reworked varies through the phases of software production. These variations are a result of different stages and processes during development, manpower-allocation, productivity and motivational factors, staff-mix and schedule pressure. I will now explain how these processes are carried out and how the effects influence the behavior above.

When a software project is established, management allocates their resources to the three major efforts undertaken in the early production phase. First management allocates 15% of the work-force to the quality assurance effort. Quality assurance covers the error detection effort. Secondly management allocates the remaining of the workforce to the software production effort, and the rework effort. The software production effort covers both the coding and the design phases of development, while the rework effort covers correction.

When the software project is underway, the production team starts to develop software. The tasks that are produced and completed are sent to the quality assurance effort, where errors are detected. While tasks are developed, errors are generated as well. Manpower allocated for quality assurance effort detects errors; these detected errors are sent to the rework effort in order to have them corrected. This is depicted in the graph above as the total number of detected errors waiting for rework increases sharply at the beginning of the project.

As long as tasks are developed and sent to quality assurance, manpower allocated for quality assurance effort checks the tasks and detects the errors that are generated during development. When errors are detected and start waiting for rework, manpower is allocated to the rework effort and rework team starts their correction of errors. Initially this work is harder at the beginning of the project as the software project is currently in the design phase of development. In the design phase errors are harder to detect due to low density. However, as design errors are fundamental errors in the software blueprint. Hence they are harder to rework, so the rework effort needed per error is much higher than the detection effort. Therefore the numbers of detected errors waiting for rework increases quickly at the first stages of development. When the software project progresses, we see that the increase in detected errors waiting for rework reduces in level. There are several key factors to this development.

The first factor for the development is the learning factor and experience. In the beginning of a project, the staff is completely new for the new project. Some of them might have experience for other projects, but in a new project they also start from scratch as no project are completely similar (Abdel-Hamid, 1991: p.44). When the project progresses, more tasks are accomplished and the product is taking shape. The staff gains knowledge of the inner workings of their software as the different design aspects are established. Therefore, error generation during production become less and when quality control identifies errors, the rework teams have more knowledge about the design and how the tasks should operate. Hence, correcting such mistakes takes less effort due to increased rework experience which leads to a higher rework rate.

The second factor in play is development and progress that is presented through percent of job actually worked. When software is developing, the project progresses and moves from design phase to coding of development. The tasks that need to be completed are now easier to accomplish. Lighter design features, and fundamental code tasks are simpler to accomplish than the core design tasks found in the initial production effort. With simpler tasks to accomplish, there are three effects influencing the detected errors waiting for rework. Simpler tasks provide less error-proneness and fewer mistakes are initially made. Secondly the increase in productivity due to learning along the project progress provides positive motivational factors. The feeling of accomplishment leads to better craftsmanship, and fewer errors. Thirdly, as progress moves the project from design to coding, the errors that need to be reworked are coding errors. These are easier to rework, and the rework rate increases as a result.

The third factor is manpower-allocation for rework effort. When the stock of detected errors waiting for rework increases at the beginning of the project, it provides management with a signal that re-allocation for rework is needed. Since design errors are harder to rework, the effort needed per error is higher. Management would then re-allocate staff to the rework effort, subsequently leading to more errors being reworked and a higher rework rate.

The fourth and final reason for the lower level of increase in detected errors waiting for rework a while after beginning of the project, lies in manpower allocation to the quality assurance effort. This relationship is vital in understanding the misperception that the amount of detected errors waiting for rework may give management.

When more errors are detected and waiting for rework, parts of the development staff is utilized for rework. This will lead to a loss of production, and management perceives a progress-delay. If this delay is severe enough, and under a schedule pressure situation, re-allocation of staff from the quality assurance effort to production effort would be a cure for this problem. This is a practice found in empirical examples provided by Abdel-Hamid, where the quality assurance effort could be suspended for weeks or even stopped indefinitely. Errors are impossible to prevent, and the size of the quality assurance team provides the potential for detecting errors. If management allocates staff from quality assurance to production, the potential for detecting errors are reduced. When the project moves from design to coding, the effort needed to detect the average error is less as well. This provides an incentive for reducing the quality assurance staff in order to boost production. The result is a situation where the detected errors awaiting rework increase very smoothly while the number of undetected errors could be increasing. This would lead to a serious misperception of the amount of actual errors.

When the software project is running along, the number of errors awaiting rework increases smoothly. However, after a time-period we see from the graph that the rate of errors waiting rework increases slowly until it reaches a peak. The reason for this development lies in the some of the factors explained above. Software productivity, motivational factors, schedule pressures and turnovers lead to an increase in the errors awaiting rework.

As the software project progresses near to the end of the project, we see that the increase in detected errors waiting for rework increase in level and reach a peak. There are some key factors to this development.

When the software project is proceeding from the design to the coding phase, the overall productivity increases. The staff is able to complete more tasks at a higher rate, and the total number of tasks controlled by the quality assurance team increases. When more tasks are being completed, the chances of making errors increase as well. This follows the logic of more attempts more misses. Secondly, a staff that has been working on the same project for a while tends to lose motivation over the course of production. Losses to motivation stems from tiredness, schedule pressures and other psychological factors. These motivational factors lead to more errors being made towards the end of the project. Schedule pressures and tiredness for example leads to a situation where staff makes bad decisions, and errors are generated at a higher rate.

The turnover factor plays in at this stage of production as well. The software industry holds one of the highest turn-over averages of all industries. When people decide to quit their job, one experienced worker is lost and an inexperienced worker takes his place. This leads to both a drop in productivity, and an increase in error-proneness of the staff. If the software project has been underestimated as well, management will need to hire more people to finish on time. This leads to additional new member of the production team and errors occur more frequently as the work-mix shifts towards a majority of inexperienced workers.

The final factor that explains this increase in detected errors waiting for rework towards the end of the project lies in quality assurance effort itself. If the quality assurance effort has been undersized or postponed for a time, the increase in the quality assurance effort will lead to an increase in detected errors. Secondly, as the software project moves from the design phase to the coding phase errors are easier to detect. When errors are easier to detect the potential for error-detection increases and more errors are detected and waiting for rework as a result.

These factors give the observed scenario of more detected errors waiting for rework, and the increase is steepest towards completion of the software project. However, as management allocate more to the rework effort, and the rework team gains experience the increase declines. The numbers of detected errors waiting for rework increases at a steady rate until the end of the project. Another factor for this steady increase is decreases in errors detection and rework rate stems. It stems from the fact that towards the end of the project most of the tasks are completed in development and manpower is allocated from production to testing. Less manpower for production means fewer tasks accomplished, and fewer tasks lead to fewer errors as well. In the end there are no more new tasks developed and no more detected errors as there is no feedback from the testing process testing process. The rework team finishes all the detected errors from previous tasks waiting for rework, and it leads to dramatic decrease in the amount of detected errors are waiting for rework at the last stages of the project and during the testing phase.

This process is governed by many factors, and it is vital for management to understand this complexity. The behavior observed here shows how project progress, manpower allocation, motivation, productivity, schedule pressure and work-force mix all influence the number of detected errors waiting for rework. Misperceptions could quickly lead to delays and problems.

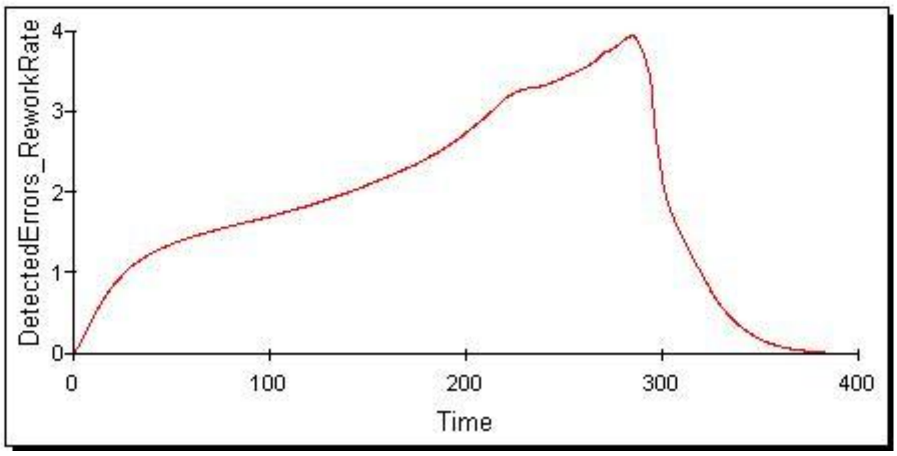


Figure 17. Detected errors rework rate

The second graph I show for the reference mode is the “Detected Errors Rework Rate” shown in figure 17. This data shows how many errors the rework teams are correcting over time. From the previous explanation we observed how project progress, motivation, productivity, experiences of workforce, schedule pressure and manpower allocation influenced the detected errors waiting for rework. From the rework rate we quickly observe that the same factors are in play here as well.

As soon as errors are detected and need to be reworked at the early stage of the project, the rework team is allocated to initialize the correction-process. From the graph we see how the rework-rate increases sharply as errors are being corrected. The rate of correction starts out strongly as the project has recently been initialized. When a project is in the starting phase, the staff allocated is all rested and ready to work. It is often an aura of excitement around a new project. Therefore the initial effort put into the earlier tasks are higher as motivation and optimism is high. These factors bring a high rate of task development and an increase in the rate of error detection. With more errors detected the stock of errors needed to be reworked increases. Since the logic of positive motivation applies for rework as well, the team is initially reworking at a higher rate. In addition there are more tasks awaiting rework at this stage as well. However, this level of effort cannot be maintained indefinitely.

After a period of time we see that the rework rate level starts to level out. The reason for this lies in the effort and the exhaustion level. In the earliest stages of development, the tasks that are completed are design tasks. Design errors are more difficult to rework, and requires far more effort per error to fix. After a few weeks of reworking difficult errors, we see that the rework staff is getting less productive. The amount of errors corrected every day declines slightly, an indication of fatigue from the rework staff. When staff feels fatigue, there are motivational losses. In the early stages optimism around a new project provides positive motivation for the crew, but with reworking errors and fatigue motivation lowers. Rework-teams are also allocated from development, meaning that every worker set to correct mistakes is unable to take part in the development of the project. This can lead to a negative association with rework. Rework could be regarded as punishment. This leads to a lower overall productivity as well. Fatigue will provide less tasks being completed, and less errors detected by quality assurance. The result is a lower rework rate.

A second factor explaining the decline here is manpower allocation for rework. At the beginning of the project when quality assurance detects errors and sends them to rework, the stock of detected errors waiting for rework, increases sharply. To accommodate this increase, manpower is allocated to rework in order to correct the waiting mistakes. When errors have been corrected at a strong rate, management is inclined to re-allocate some of the members back to production. It is of high priority for management to keep production running at peak efficiency to complete the task on schedule. Hence, allocating more personnel to production is a common policy whenever it is possible. Additionally, when the project moves towards coding, the effort needed per error is reduced. Management allocates less manpower to this effort, as a result. This is part of the explanation for the relative slow increase in rework-rate even though the project is moving from design to coding.

The third factor here is turnover. If there has been a cycle of turnovers after the initial start of the project, the rework staff has lost experienced members. Experienced members are valuable, as their expertise provide high productivity. Secondly, with turnovers the communication-overheads increase as well since the members of the team have new people to learn to communicate with. Continuous loss of partners in a working-team leads to motivational losses, as workers feel a sense of instability in the workplace.

After a while, the rework-rate picks up speed again. In the graph we see an in increase in the curve providing the second strongest rework-rate increase. The first explanation lies in the software production efficiency. When more tasks are being completed, the project moves from design to coding. In the coding phase, errors that need to be reworked are much simpler to handle. This allows the same amount of staff to rework more errors per day, and the efficiency starts to increase. In addition, when a project is moving from design to coding the staff has gained more knowledge of the inner workings of the program. This knowledge allows them to be more effective in correcting errors and the rework rate increases as a result. Due to the same logic, the production rate of tasks and error detection rate from quality assurance provide more tasks for the rework staff to correct. Hence, their increased capacity is needed.

The second explanation lies in manpower allocation. We saw that the quality assurance effort detected more errors towards the end of the project increasing the stock of detected errors waiting to be reworked. The managerial response to such a situation is more manpower for the rework effort. This factor is also influencing the increase in the rework rate here.

Finally the amount of schedule pressures increases the rework rate for a period of time. As the project closes to the finish-line the schedule pressures increase. Increase schedule pressures provide a feeling of urgency in the rework team, and they will cut slack time and increase their productivity. This translates into a higher rate of rework as we can observe from the graph above.

This increase is carried on for a period of time, before the level of rework diminishes slightly again. Reworking coding errors are easier, but not motivational. Reworking coding errors could quickly lead to a feeling of doing janitorial tasks. Secondly an increase in schedule pressure leads to an increase in fatigue and lower productivity. This slump-period is much shorter however as the production of software is close to the end-stage. More tasks are developed and completed, so less manpower is needed for software production. This translates to more manpower available for rework. Quality assurance personnel and software development teams are being sent gradually to testing. However, when software production halts, there is a delay between no new tasks developed and all tasks are quality assured and errors are detected and reworked. Hence management allocates more effort to rework as well as testing. This results in a final peak in rework efficiency at the end of the project. At the end of the project during testing phase, there are no more detected errors sent back to production due to the lack of feedback between rework and testing. Hence there are less and less detected errors to rework, and the rework rate drops after some time to zero. Management allocates more people from rework to testing as the number of detected errors waiting for rework declines. This explains the slope-decrease towards zero as the rework team is gradually reduced.

Chapter 6: Dynamic hypothesis and model structure

6-0 Introduction

In my previous chapters I have presented the concept of the software crisis. My motivation is to improve overall software quality during the software production project. My literature has established how quality can only be achieved through an improved quality of production. I have presented Abdel-Hamid's base model, which a pioneering effort in software production. This model, combined with the literature, has helped me construct entries for improvement. This chapter contains my major contributions, explaining how enhancements to the base model will apply more realism to it first and then enable us to achieve higher software quality. The system dynamic methods, as outlined, provide the valuable tools deployed here, both causal loop diagrams and stock and flow diagrams. In the previous chapter, I presented three research questions, which will be investigated through the enhancements presented in this chapter.

Through this process I will highlight the important modifications in the three central processes for software quality (quality assurance, rework and testing), and incorporate the client as an actor in the testing process as a potential policy. The different considerations and enhancements for each sector will be clarified in this chapter. The dynamic hypothesis for every enhancement will be presented as a verbal description, followed by a CLD that provide the logic and dynamic behind it. At the end the model structure as a stock and flow diagram will be provided and central main parts and variables will be explained.

6.1 Model enhancement Boundary

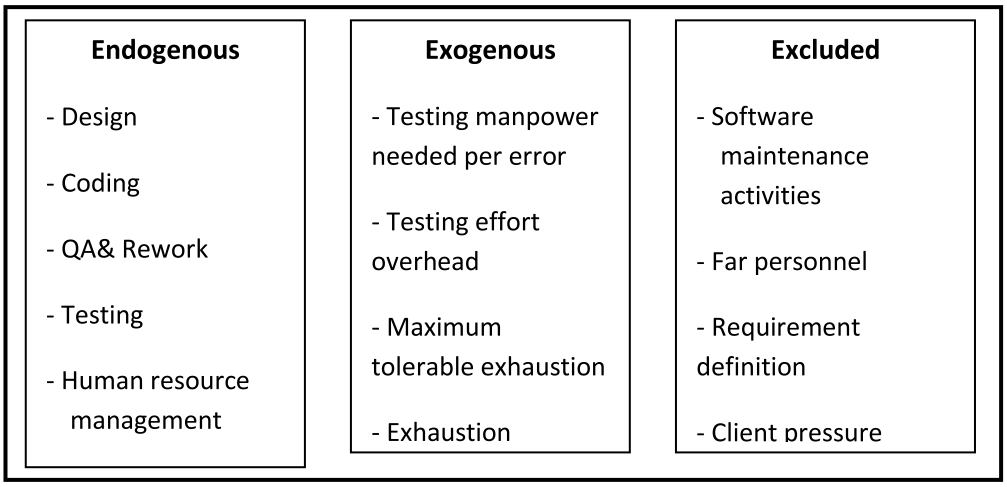
Since models are a simplified description of real-world processes it is important to identify their boundaries. Model boundaries provide the scope and applicability of models within the phenomena they are created to represent. Unless the boundaries are expressed, a model could technically attempt to recreate and incorporate all observed variables in the real world system. This would constitute a grave problem as some simplifications are necessary in order to establish important relationships between variables. With no model boundary the density of variables and details would drown out the causes for observed behaviour and policy-opportunities. Hence, a model needs explicit boundaries (Sterman, 2003).

In system dynamics, behaviour and behaviour patterns emerge from endogenous variables. Abdel-Hamid's model holds some clear boundaries both in relation to size and process. I will now introduce the main boundaries of the base-model and provide a table were the factors included and excluded are represented.

The first boundary to the base-model is size of project. The model is based on a medium-ranged software project as defined by Jones. The medium ranged project contains between 16 KDSI and 64 KDSI. Projects that are smaller or larger in size will not be considered applicable for analysis in the base-model (Abdel-Hamid, 1991: p.46).

The second boundary of the base-model is a clarification of processes that are included and excluded in the endeavour. The goal of the model is to identify the causal relationships between production processes when a software project has been initiated until the final stage of testing. Hence, the base-model assumes that pre-planning activities and budgets have been determined. The process of determining the software requirements, allocating funds and workforce capacities are therefore not included. In the other end of the process, the efforts that are put into the product post-development are not included as well. Post-production maintenance or redesign for example is outside the base-model scope.

The endogenous activities that are included in the model, all constitutes the defined production activities. Design and Coding, Quality Assurance and Rework and Testing are example of activities that are included. The table below shows the endogenous and excluded variables.



Endogenous	Exogenous	Excluded
- Design	- Testing manpower needed per error	- Software maintenance activities
- Coding	- Testing effort overhead	- Far personnel
- QA& Rework	- Maximum tolerable exhaustion	- Requirement definition
- Testing	- Exhaustion	- Client pressure
- Human resource management		

Figure 18. The endogenous, exogenous and excluded processes of the base model

In the table above we see the endogenous processes of the model and the excluded processes according to the boundary. The focus is to identify the causal relationship and behaviour-patterns between actors within the production cycle itself. Hence, far personnel are excluded. Preliminary planning requirements are defined and project-budgets constructed are also excluded, as these processes are outside the main development process. From the table above we see that the client has been excluded from the model itself. In the model, one assumes that client demands are constant and that client pressures are not subjected to the software developing team. Hence, clients are regarded as agents outside the base-model. These are the main boundaries of the base-model (Abdel-Hamid, 1991: p.20).

When I provide the enhancements to the model, I will expand the original model boundaries to fathom the client as an exogenous force. There are two reasons for this choice. Firstly my aim is to provide the model with added realism in order to provide better software quality. From the literature we learn that software quality and client satisfaction are two related factors. With the development of acceptance testing and end-user reviewing, the client should be included as a factor here.

Hence the model will now include one review process that includes the client, and the new model boundary will run vertical between the manpower allocated by management, and manpower allocated by the client to client-reviewing. In this fashion I expand the model boundaries in order to capture an important base for increased software quality and enhanced quality policies. The table below shows the new model-boundaries for the enhanced model.

Endogenous	Exogenous	Excluded
- Design	- Client personnel for client reviewing	- Software maintenance activities
- Coding	-Client Manpower effort to review	- Far personnel
- QA& Rework	- Client manpower efficiency	- Requirement definition
- Testing		- Client pressure
-Client Testing		- Changes in client demands
- Human resource management		

Figure 19. The endogenous, exogenous and excluded processes of the enhanced model

Note that the client pressures and changes in demands are still excluded here. The background for changes in client perception in this model rests on their experience from client-testing and not from other sources.

6.2 Time Horizon

It is important to select a sufficient time horizon for my model simulations. This varies from software project to software project, depending on size, complexity and scope. Since we are running an enhanced version of the model that is 64 KDSI in size, it is natural to select a time frame that covers the typical amount of days spent on a medium-sized software project. For the base-model Abdel-Hamid provided a time frame for such projects from the DE-A example. The DE-A project ran for about 407, so the time-horizon chosen for the original model was 430 days.

For the new enhanced model we need to take into consideration that the project most likely will run for a longer time period due to the added processes and policy structures. Secondly, as the average delay for a software project is 33% I need to set a time-interval that is sufficient enough to capture all activities in the software project from start to finish. Therefore my time-horizon will be 600 days, as I deem this a sufficient time-period for capturing all the software project activities and that all the tasks are completed.

6.3 The rework process

The first process that will be enhanced in this thesis is the rework process. Upon further investigation it is apparent that there is room for improvements that will provide more realism for this process. In the rework process there are two areas where key concepts are absent in the original model that have important influence.

The first area is the effort needed during the rework process. The rate of rework in the original model is influenced by the rework effort needed per average error during the rework process. This is determined by the nominal rework manpower needed per error due to error type, and the multiplier due to communication and overhead losses. These two variables do indeed affect the rework rate; however this leaves other influences absent. In my research, error type, error density and software development productivity will influence the rework effort needed per average error.

The second area is the rate of bad fixes generation during the rework process. The error type, error density, schedule pressure and fraction of experienced workforce in the project will have impact on the rate of bad fixes generation. Bad fixes are a constant source for increased rework, as bad fixes may constitute as much as 20% of the errors occurring in the system (Jones, 2007). Abdel-Hamid utilizes in his base model a constant for the bad fixes generation. He argues that the empirical data provide a range of bad fixes between 6 to 10 per cent, and that he holds the rate constant at 7.5%, which is the industry average. From recent literature, we learn that the industry average is indeed 7%, however, truly successful software projects report a bad fixes generation rate of 2%. In very unsuccessful projects, it may be as high as 25% (Jones & Bonsignour, 2012).

Considering the facts above, these two areas of the rework process are in need for enhancements. These enhancements will now be presented in detail.

6.3.1 Manpower effort for rework

From the base model we learned that error density holds impact on the quality assurance effort. Through the production, the software development process moves from the design phase to the coding phase. In the design phase the effort to detect an average error is higher than in coding. This is due to the nature of design errors and code errors discussed earlier in chapter 4.

Design errors are more subtle, and therefore harder to detect. Coding errors are easier to detect. They are in nature more obvious errors, and in the coding process they tend to be more plentiful. When production has moved from design phase to coding phase the error density will be higher. Higher density leads to simpler detection as obvious errors occur. Secondly a pattern of obvious errors may lead to the revelation of subtle errors. Therefore an increase in density provides more information on hidden errors as well (Abdel-Hamid, 1991: pp.105-106).

The logic behind error density and quality assurance process is logical and applicable to the rework process as well. The quality assurance team identifies errors; they do not fix them themselves. Detected errors are gathered and await rework by the rework team. The rework team is members of production staff that are assigned to this process by management.

Reworking errors follow nearly the same logic as detecting them. Design errors are indeed revealed by the quality assurance team, but the solution to correct the errors must be applied by the rework team. Therefore, design errors are harder to rework and the average effort needed per such errors is greater than for coding errors.

The error density also provides the same pattern of errors for rework as for quality assurance. When there are several obvious errors found, the pattern among them may reveal a subtle design error causing the effect. Design errors are not passive, and they generate code-errors. Therefore the importance of error density will be applied to Abdel-Hamid’s model in order to enhance the realism and strengthen overall software product quality. The logic is provided in the CLD below.

In these new CLDs provided in this chapter I directly link the new loops and variables to the based ones presented in chapter 4 and then zoom in on the new loops created after applying the enhancements to explain the loops in detail. The variables and arrows in purple are the new additions that my enhancements provide. Black variables and dark blue arrows are the original variables and relations inside the subsystem. Light blue variables and arrows are the original across subsystem variables and relations.

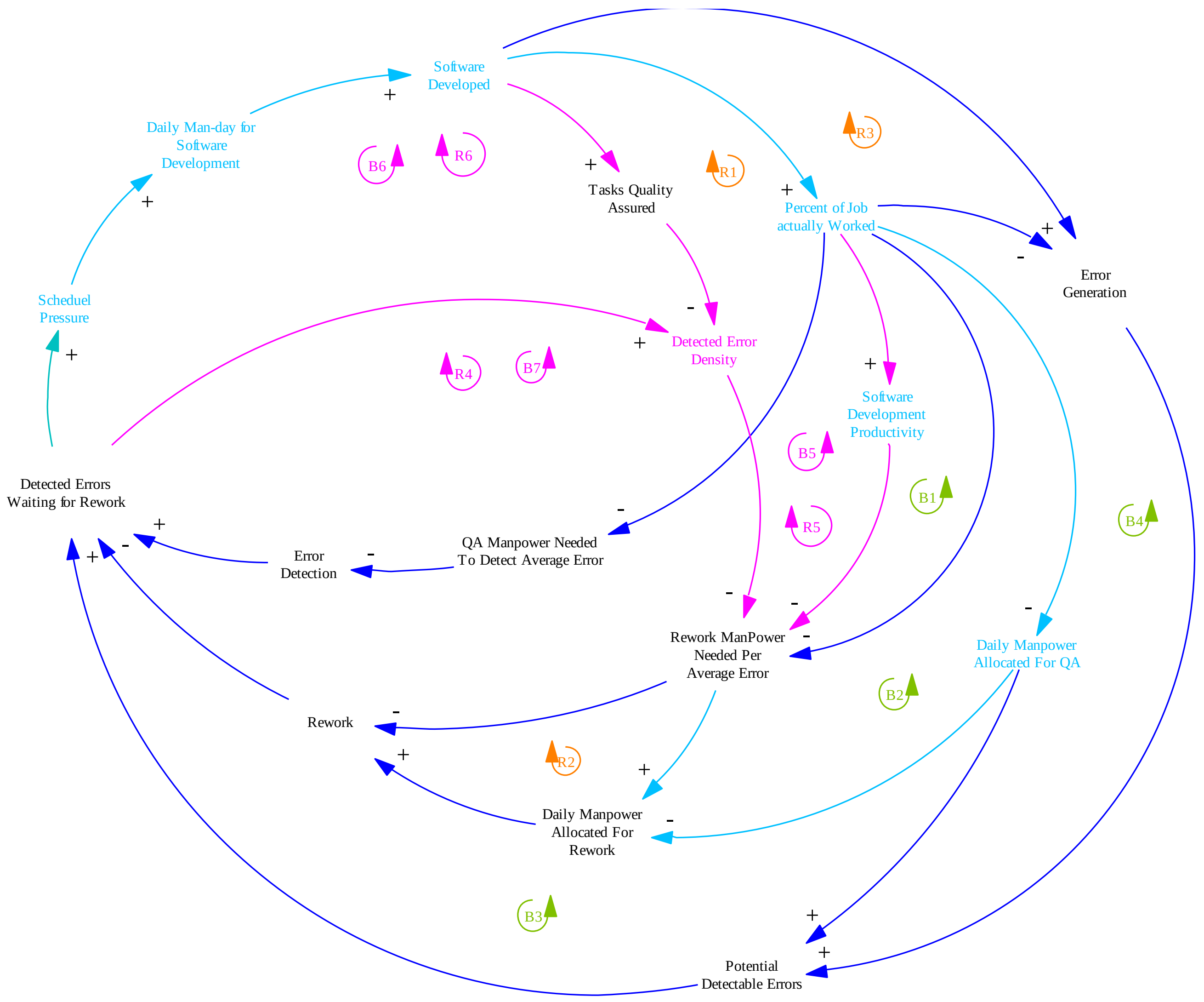

Software Developed
Daily Man-day for Software Development
Tasks Quality Assured
Percent of Job actually Worked
Error Generation
Scheduel Pressure
Detected Error Density
Software Development Productivity
Detected Errors Waiting for Rework
QA Manpower Needed To Detect Average Error
Error Detection
Rework ManPower Needed Per Average Error
Daily Manpower Allocated For QA
Rework
Daily Manpower Allocated For Rework
Potential Detectable Errors
B6
R6
R1
R3
R4
B7
B5
R5
B1
B4
B2
R2
B3

Figure 20. Quality assurance and rework subsector CLD,

With new loops provided by enhancing the effort for the rework process

From the previous diagram is shown in figure 20, we see that detected error density enhancements have been added, and six new loops emerge in the subsystem from this addition. Now I will enter the sub-category and explain the new loops. In this CLD I have removed the other loops in order to filter the relationships that are essential to my enhancements.

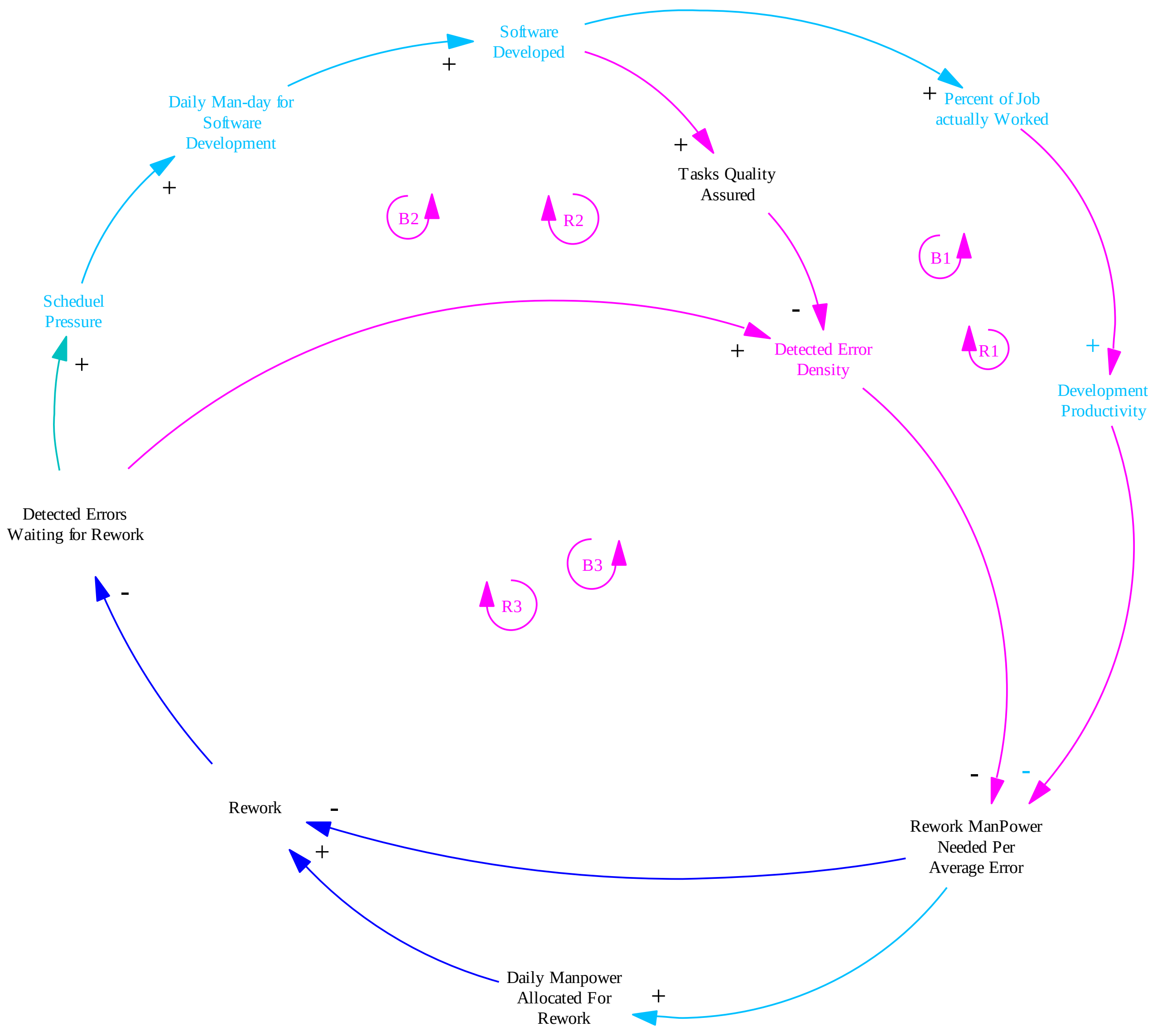

Software Developed
Daily Man-day for Software Development
Percent of Job actually Worked
Tasks Quality Assured
Scheduel Pressure
Detected Error Density
Development Productivity
Detected Errors Waiting for Rework
Rework
Rework ManPower Needed Per Average Error
Daily Manpower Allocated For Rework
B1
B2
B3
R1
R2
R3
+
-

Figure 21. New loops are provided by enhancing the effort for rework process

R1: The first reinforcing loop of this new system concerns the production of software, percent of job actually work, development productivity and the rework effort and process.

When software is being produced, the percent of job actually worked increases. The percentage of job done indicates where in the software process the project is currently situated. When more percentage of the work has been completed the project moves from the design phase to the coding phase. Through the design phase, the development team constructed the software’s design frame. When the team work and complete tasks, they learn about the software system they are creating, and how it operates. When the project moves along this learning and knowledge allows the production-rate to increase. The teams work more efficiently due to learning, and tasks are being completed at a higher rate.

This in turn influences the rework rate as well. The rework effort is carried out by manpower allocated from the development team. When they have experience with the software and learned, the correction solutions to errors are more apparent as well. Hence, the rework team solves problems at a higher rate. The effort needed per average error decreases accordingly. Hence the relationship between the productivity and effort per average error is negative.

The rework manpower needed per average error influences the daily manpower allocated for rework. When the effort to correct an error lowers, so will the daily allocated manpower to rework lower. With a lower level of daily manpower allocated to rework, the rework rate is decreasing intact with the reduction. Decreased rework rate increases the detected errors waiting for rework. The increase in errors awaiting rework leads to increase in schedule pressures, and management re-calculates and increases the daily manpower for software production as a result. The loop closes as the adjustments increases the software development rate.

R2: The second new reinforcing loop of the system concerns the production of software and the rework process and effort in conjunction with detected error density.

Software is being developed, and the quality assurance effort provides quality assured tasks. The density level of the errors detected decreases, and makes the rework process harder to perform. The low density influences the rework manpower needed per average error. When density is high, the pattern of errors provides information on how to best fix them. Secondly, a high error pattern consists of code-errors that are easier to rework. When density is low however, there is no apparent pattern emerging from the errors indicating how to best fix them. Secondly, the errors are primarily design-errors that are harder to rework.

With an increase in the manpower needed to rework an average error, the rework rate is reduced as a result. The reduction in rework rate causes a situation where the number of errors waiting for rework increases. The increase in errors awaiting rework leads to increase in schedule pressures, and management re-calculates and increases the daily manpower for software production as a result. The loop closes as the adjustments increases the software development rate.

B1: The first balancing loop of the system concerns the production of software, percent of job actually work, development productivity and the rework process and effort.

When software is being produced, the percent of job actually worked increases. The percentage of job done indicates where in the software process the project is currently situated. When more percentage of the work has been completed the project moves from the design phase to the coding phase. Through the design phase, the development team constructed the software's design frame. When the team work and complete tasks they learn about the software system they are creating, and how it operates. When the project moves along this learning and knowledge allows the production-rate to increase. The teams work more efficiently due to increase in productivity, and tasks are being completed at a higher rate. This in turn influences the rework rate as well.

The rework effort is carried out by manpower allocated from the development team. When they have experience with the software and learned, the correction solutions to errors are more apparent as well. Hence, the rework team solves problems at a higher rate. The rework effort needed per average error decreases accordingly. So the relationship between the productivity rate and effort per average error is negative.

The rework manpower needed per average error influences the rework rate. When less rework manpower is needed per average error, the rate of rework is increasing as a result. This leads to a subsequent decrease in errors waiting for rework. So the relationship between these three variables is negative. Less errors waiting for rework leads to an decrease in schedule pressure, and management decrease the daily man-day for software development as a response. The loop is closed as the adjustment decrease the software developed rate.

B2: The second balancing loop concerns the relationship between the production of software and the rework process and effort in conjunction with detected error density.

When software is being produced, the quality assurance team will start their work. This yield tasks that have been quality assured. When the number of tasks quality assured increases, the total density of detected errors decreases. Hence, the relationship between the two variables is negative. When the error density is low, the rework manpower needed per average error increases providing a second negative relationship. When more manpower is needed per average error, the daily manpower allocated to rework will be increased. The rate of rework depends on the daily manpower allocated to rework, and with an increase the rate of rework increases accordingly. When the rework process commences, the number of tasks waiting for rework diminishes as the rework team reworks them. The decrease in amount of detected errors awaiting rework influences schedule pressure. When rework is underway, and the number of tasks waiting for rework diminishes and schedule pressures are reduced. The reduction in schedule pressure decreases the daily manpower needed for software production and the loops is closed when these managerial adjustments decreases the software developed.

B3: The third balancing loop of the system concerns the relationship between detected error density and rework effort and process. When the total density of errors increase, the error patterns are easier to identify. The result is a decrease in the rework effort needed per average error. This increases the rework rate, and as a result fewer errors await rework. The loop is closed when fewer errors are waiting to be reworked and the error density decreases.

R3: The final reinforcing loop in this system concerns the relationship between detected error density and rework effort and process. When the error density increases, the project is moving from the design phase to the coding phase. Increased error density translates into more coding errors and observable error patterns that make rework easier. Hence, when error density increases, the average manpower needed per error decreases. With a decrease in manpower needed per error, the daily manpower allocated to the rework effort is adjusted downwards by management. With less daily manpower allocated for rework, the rework-rate decrease as well. This leads to an increase in detected errors waiting for rework, and the detected error density increases as a result. This closes the final reinforcing loop.

The causal loop diagram is an important tool for identifying the logical links between variables inside a dynamic system. The diagram is limited, and I will therefore present an enhanced stock and flow diagram that will identify all the essential variables and factors to the influence of error density on rework effort and process. This enhanced stock and flow diagram will be part of the large overall model connecting all the subsystems. For the sake of clarity, the stock and flows have been extracted from the sub-system.

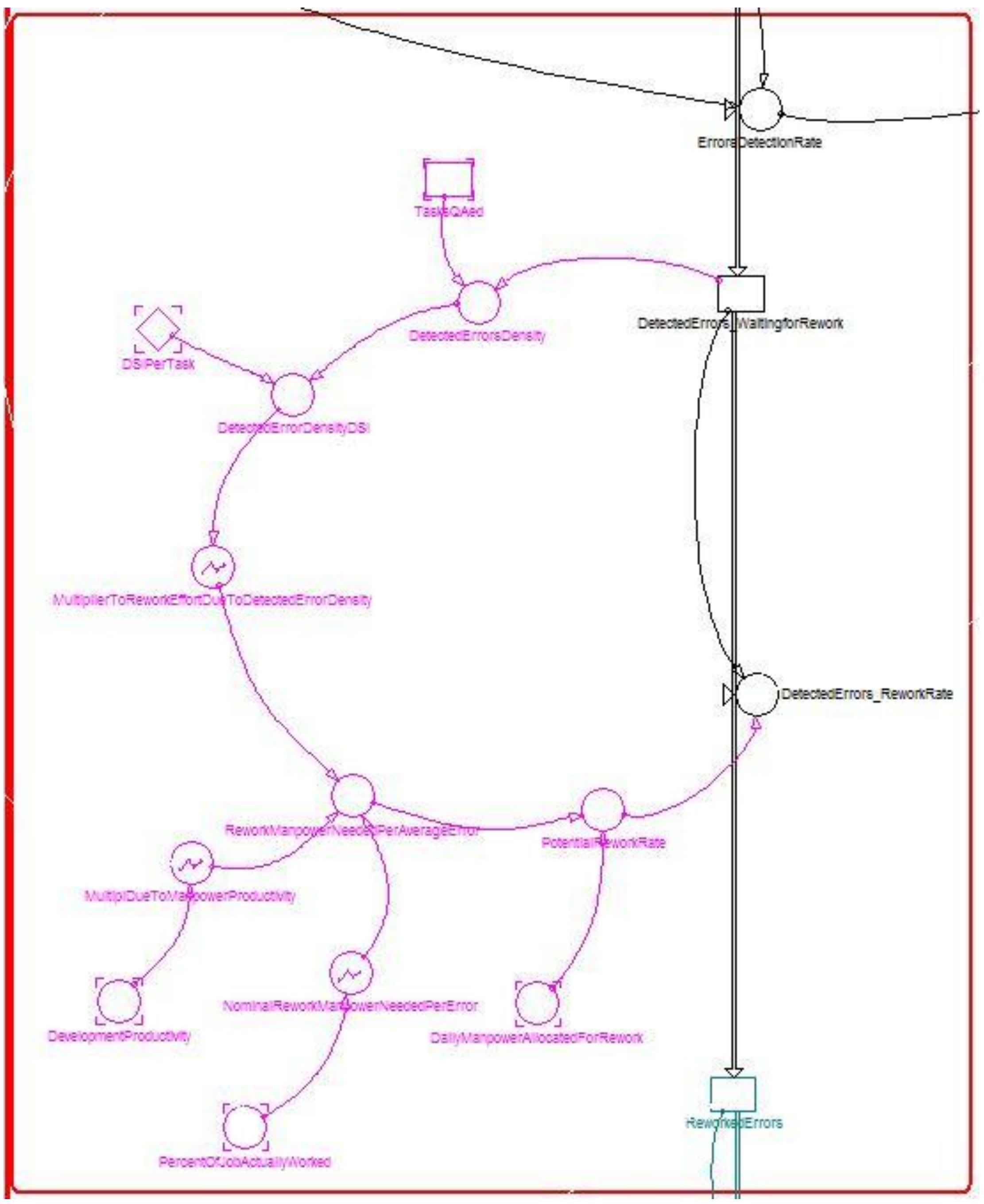
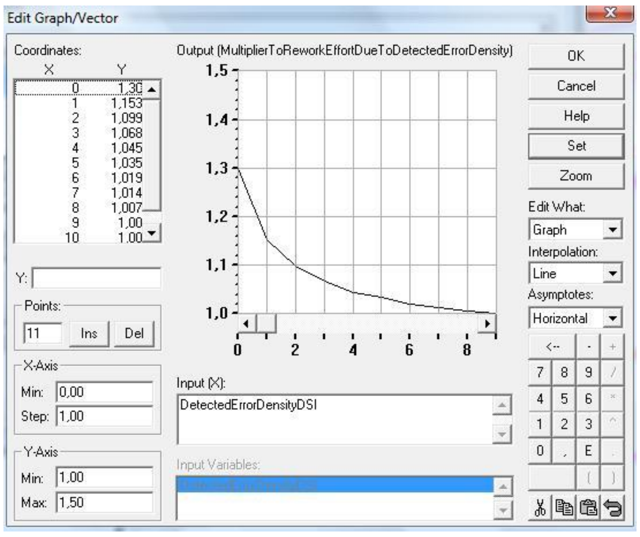

ErrorsDetectionRate
TasksQAed
DetectedErrors_WaitingforRework
DetectedErrorsDensity
DSIPerTask
DetectedErrorDensityDSI
MultiplierToReworkEffortDueToDetectedErrorDensity
DetectedErrors_ReworkRate
ReworkManpowerNeededPerAverageError
PotentialReworkRate
MultiplDueToManpowerProductivity
NominalReworkManpowerNeededPerError
DailyManpowerAllocatedForRework
DevelopmentProductivity
ReworkedErrors
PercentOfJobActuallyWorked

Figure 22. Stock and flow diagram is provided by enhancing the effort for rework process

The first factor that affects the rework effort is the detected errors density. The rework process begins with two important stocks. When quality assurance has started its effort, tasks are being controlled and errors are detected and cumulate as detected errors waiting for rework. The detected errors waiting for rework divided by the cumulative tasks quality assured provides the detected error density level. This level is transformed into an errors density level per DSI, as tasks are measured in DSIs.

The detected errors density influences the rework effort through the "multiplier to rework effort due to detected errors density". From the literature, and interviews with software engineers we know that low density provides more rework effort per average error needed rework. This is due to the nature of design errors and the absence of code errors to reveal a fix-pattern (Jones, 2007). In my enhancement the multiplier is a graph-function where the initial density point at zero and the rework effort multiplier is set at 1.3. This is consistent with empirical data that suggests design errors that are less in number but provides more time and effort per average error (Jones & Bonsignour, 2012). From their empirical data, the number of hours spent on reworking certain types of errors ranging from code to design is presented (Jones & Bonsignour, 2012: p.263). Design errors need on average about 2 hours more to be reworked than coding errors. Additionally, the most severe design-errors require more effort than the most severe coding errors. Severe design errors may require 4 hours to repair per defect compared to 2.5 hours per defect for severe coding errors (Jones & Bonsignour, 2012: p.236).

Taking the figures into consideration and calculating for the averages I find that a multiplier of 1.3 is appropriate in this case, and the graph below depicts the relationship of the multiplier. The difference in hours and so relatively rework efforts, needed from severe to less severe errors make the graph descend rapidly until leveling off towards zero. The difference in hours diminishes between light design and coding errors providing a flatter curve between these errors.



Edit Graph/Vector

Coordinates:

X	Y
0	1,30
1	1,153
2	1,099
3	1,068
4	1,045
5	1,035
6	1,019
7	1,014
8	1,007
9	1,00
10	1,00

Output (MultiplierToReworkEffortDueToDetectedErrorDensity)

Points: 11

X-Axis Min: 0,00 Step: 1,00

Y-Axis Min: 1,00 Max: 1,50

Input (X): DetectedErrorDensityDSI

Edit What: Graph

Interpolation: Line

Asymptotes: Horizontal

Figure 23. Multiplier to rework effort due to detected errors density

When error density increases, the rework effort to correct the average error lowers subsequently until, in very high detected errors density, it reaches the value one and holds no effect. With a high rate of error density, the majority of errors are coding errors. The coding errors also reveal a possible error-pattern allowing increased knowledge of underlying design errors. The multiplier to rework effort due to detected errors density provides in turn data to the rework manpower needed to correct an average error.

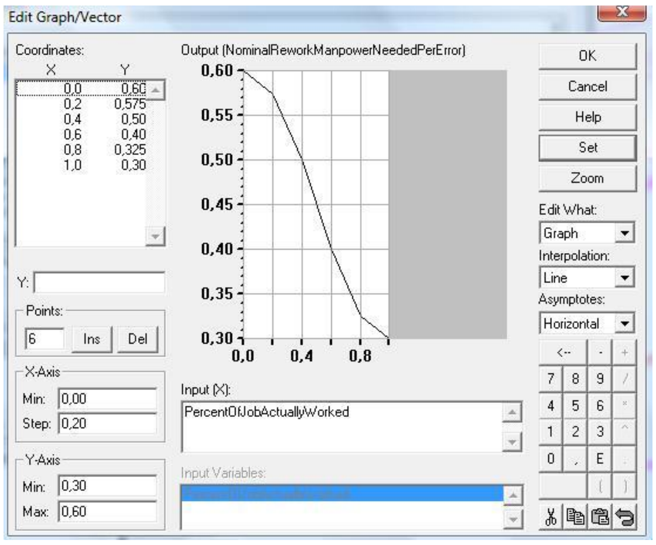
More density of errors means more typographical and small errors. Less density of errors means logical and big technical errors. When a subtle error is detected, its awareness is known. The effort to find out the source of the error and how to rework it remains. It is not only "here is the error" but "Why is it here?" However, the reason that the multiplier for the quality assurance effort is higher than the rework effort rests on the nature of the effort. When error density is low, the quality assurance effort is very hard. Quality assurance is trying to find something that is subtle in a vast empty room. High density reveals error-patterns that can give the design-error away, when no such pattern exist, it is like finding the needle in the haystack. Rework, on the other hand, already knows where the error is as quality assurance detected it, and the effort is subsequently lower in the initial effort at low density.

The second factor that affects the rework efforts is development productivity. The rework effort needed to correct the average errors is also influenced as the quality assurance effort by the productivity of the staff. The staffs are assigned to rework errors as they are detected by the quality assurance team. Their productivity is measured as potential productivity minus the effect of motivational losses and communication overheads.

Recall that rework staff is allocated from the development side. Hence, the productivity factors influencing the rate of software development applies here as well. The three factors that are affecting the production productivity are fraction of the experience work force, the total work force level and the percentage of job actually worked. An experienced work force has a better understanding of how rework should be carried out. Percentage of job actually worked indicates that the project is progressing. The production staff is learning due to completed tasks and the clearer framework of the project as a whole. It also indicates where in the development phase the project is, while it’s progressing from design to coding.

As explained before coding errors are easier than design errors to rework. The total work force level provides the amount of communication overheads. The more people on the job, the more time is spent on inter-communication that lowers the productivity. In this manner we apply the productivity itself as a determinant rather than a multiplier due to communication and overhead losses. This enables us to apply the important key effects that influence the effort and provide more realism to the model.

The third factor that influences rework effort is already applied in base model and it is the nominal rework manpower needed per error due to error type. This is a graph-function as well, that show how the average manpower needed per error differs by error type and between the design and coding phase. The design errors that are detected in the earlier stages are more costly to rework as well. When the percentage of job actually worked increases, the project moves from design phase to coding phase as indicated in chapter 4. The nominal rework man-day needed per average error in the design phase is 0.6 while at the end of coding holds the value 0.3 (Abdel-Hamid, 1991: p.107). The graph below shows this relationship:


Edit Graph/Vector

Coordinates:

X	Y
0,0	0,60
0,2	0,575
0,4	0,50
0,6	0,40
0,8	0,325
1,0	0,30

Output (NominalReworkManpowerNeededPerError)

Points: 6

X-Axis Min: 0,00 Step: 0,20

Y-Axis Min: 0,30 Max: 0,60

Input (X): PercentOfJobActuallyWorked

Edit What: Graph

Interpolation: Line

Asymptotes: Horizontal

Figure 24. Nominal rework effort due to error type

The rework manpower needed per average error is then a function of the nominal rework manpower needed per average error and the multiplier to rework due to error density and the development productivity. The variable is formulated as follows:

*Rework manpower needed per average errors= Nominal rework manpower needed per error * Multiplier To rework effort due to detected error density * (1 / Development productivity)*

From the rework manpower needed per average error, and the daily manpower allocated for rework we find the potential rework rate. This variable then influences the outflow of the stock of detected errors waiting for rework. The rework rate is decided by the potential rework rate and the numbers of detected errors awaiting rework divided by a time step. In some cases, the number of detected errors is below the capacity of the rework force for a given period of time. To prevent negative stock, Abdel-Hamid's formulation needs to be modified. His formulation was:

Errors Rework Rate = Daily Manpower Allocated For Rework / Rework Manpower Needed Per Average Error, while mine is: MIN (Potential Rework Rate, Detected Errors Waiting for Rework / TIMESTEP).

Where potential rework rate has the same formula as the rework rate in the base-model. By applying this, I have a logical control on the outflow.

6.3.2 Bad fixes generation

Bad fixes occur when the rework effort erroneously attempt to fix an observed error. The problem of bad fixes is highly documented in the software development literature (Abdel-Hamid, 1991: p.108). Bad fixes may stem from a correction based on faulty analysis, that the fix only works locally, or that correction is accomplished by the creation of a new error.

In the base model this relationship was depicted as a relationship between the rework rate, and an assumed percent of bad fixes. Abdel-Hamid admits that the fraction of bad fixes has not been determined by any published data, but that some results indicate a percent of bad fixes between 6.5% and 10% (Abdel-Hamid, 1991: p.108).

From the software development production logic however, we have seen how experience of staff, overheads, error density, error type and schedule pressure influence the staff in their development efforts. Bad fixes are generated when the rework staff erroneously attempts to fix a problem. This happens as a result of faulty analysis or by creating a fix that leads to new errors in other parts of the code.

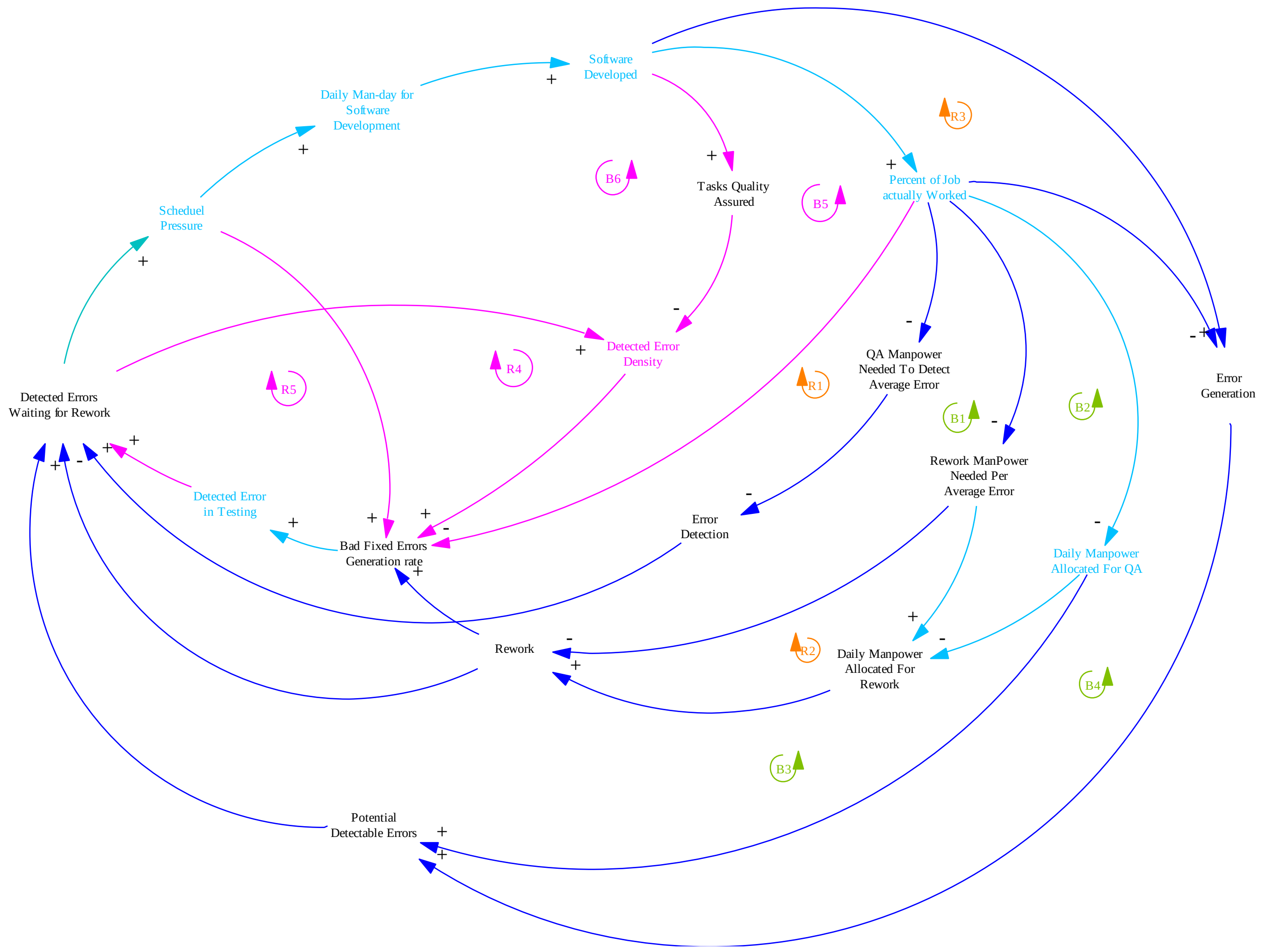
Just as for the production process, there are some factors that determine the rate of errors made in the rework process. Bad fixes made while attempting to fix errors, occur due to five factors: error type, Detected error density, staff productivity due to learning, the experienced fraction of staff and schedule pressure in the project.

As the project progress, the chance of making a mistake decreases due to increased productivity and learning. It is also due to the nature of the errors. In the earlier stages of the project, in the design phase, errors committed are harder to fix. The coding errors occurring in the coding phase later are easier to fix. Coding errors are simpler to correct and holds relatively little challenge to an experienced programmer.

However, as the project is moving from design to coding, the error density increases as well. More tasks are completed at a higher rate, and thus the chances of mistakes being made also increase. Coding errors are easier to fix than design errors but the density of errors is greater. When error density increases the likelihood of making a bad fix increases by the magnitude alone.

Secondly coding-errors are low in challenge but high in magnitude. This constitutes a motivation-loss for the programmers that are set to do this job. In a working environment positive motivational factors include the opportunity to expand one's knowledge and improve skills. By being set to remove simple errors, the task provided is under-stimulating. Rework is felt like a janitor-service and motivation drops. When a worker is unmotivated, the likelihood of making a mistake by cutting corners or lack of focus increases. Schedule pressures also play a vital role here, as stress leads to more errors. When workers are hasting to finish a job, the speed and lack of calm focus increase the error-proneness of the rework team. Cutting corners and making quick fixes that seem to work in order to finish on time will lead to more bad fixes.

Having experienced staff is a big influence to the bad fix error-generation. When the staff consists of workers that have been employed with the company for a while, and have experience with previous software projects they will be an important asset for the rework effort. Experienced personnel can draw on their knowledge from previous rework-efforts and how codes should be fixed. The fact that they know the company structure and procedures allow them to work more effectively. Workers that have been working together in teams before communicate better between each other cutting communication losses as well. The experience allows the worker to work efficiently and to maintain focus when schedule pressure rises, leading to less bad fixes during any scenario compared to completely new workers. The CLD below shows this relationship:



Software Developed
Daily Man-day for Software Development
Scheduel Pressure
Tasks Quality Assured
Percent of Job actually Worked
Detected Error Density
QA Manpower Needed To Detect Average Error
Error Generation
Detected Errors Waiting for Rework
Detected Error in Testing
Bad Fixed Errors Generation rate
Error Detection
Rework ManPower Needed Per Average Error
Daily Manpower Allocated For QA
Rework
Daily Manpower Allocated For Rework
Potential Detectable Errors
R1
R2
R3
R4
R5
B1
B2
B3
B4
B5
B6

Figure 25. Quality assurance and rework subsector CLD,

With new loops provided by enhancing the bad fix generation rate

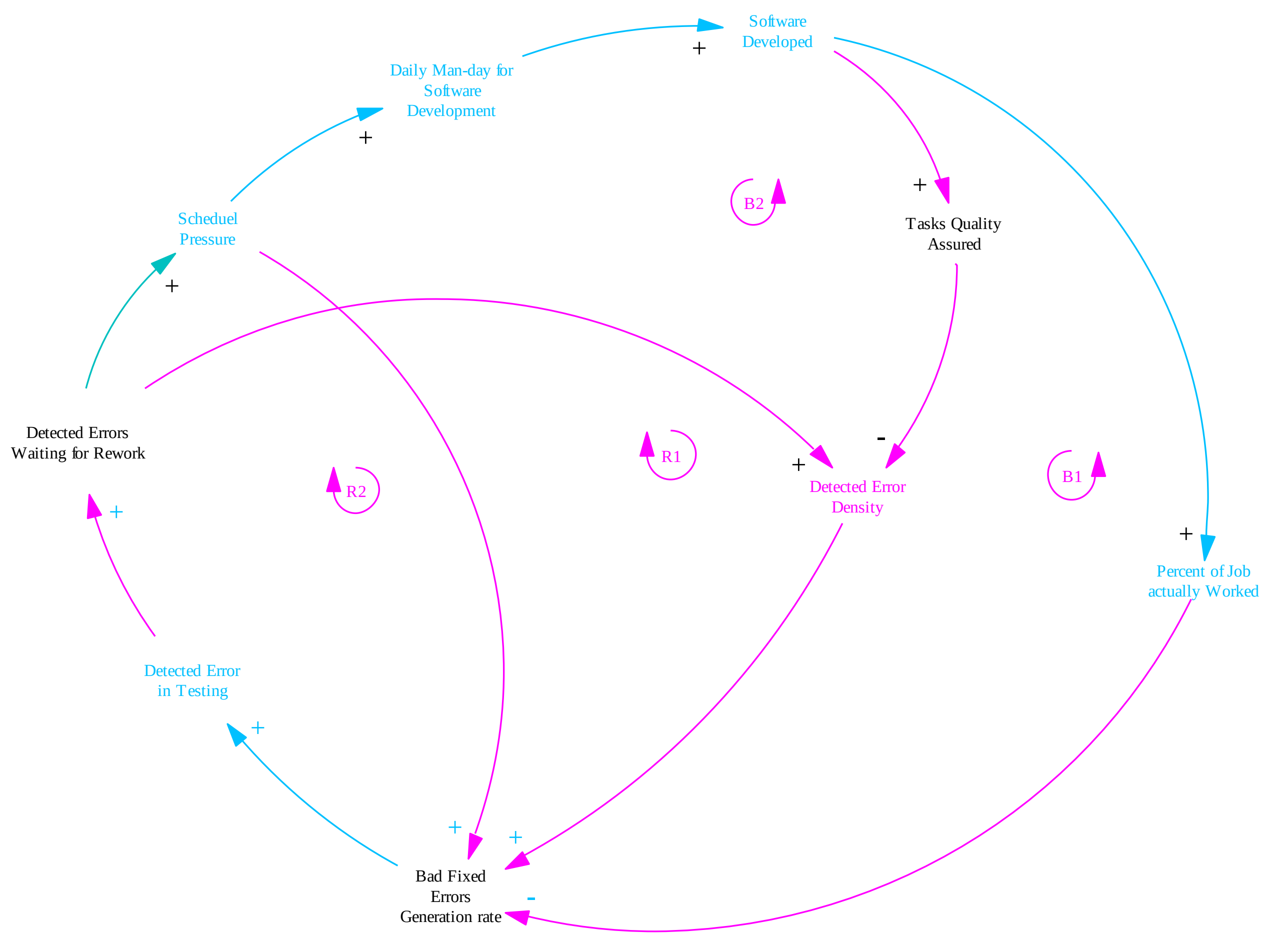

Software Developed
Daily Man-day for Software Development
Scheduel Pressure
Tasks Quality Assured
B2
R1
R2
B1
Detected Errors Waiting for Rework
Detected Error Density
Percent of Job actually Worked
Detected Error in Testing
Bad Fixed Errors Generation rate
+
-

Figure 26. New loops are provided by enhancing the bad fixes generation rate

B1: The first balancing loop of the system is related to the development of software, the percent of the job done, bad fixed error generation rate, detection of errors in testing and schedule pressure.

When software is developed, the percentage of the work actually done increase. When the project has accomplished 50% of the assigned development tasks the model assumes the project has moved from design phase into the coding phase. When the process is in the coding phase, the generations of bad fixes decrease. With mostly coding errors, the obvious nature of such errors makes them easier to fix. Hence the relationship between the two variables is negative.

When bad fixes are not committed, the number of errors detected in testing decrease. When the testing-effort is not finding new errors due to the lack of bad fixes, the detected errors waiting for rework decrease as well. This leads to a decreasing of schedule pressures, and management may decrease the calculated daily man-days needed for software development. The loop is closed as the adjustment decrease the rate of software development.

B2: The second balancing loop identifies the relationship between the software production, detected error density, bad fixes, errors detection in testing and schedule pressure.

When software is being developed, the quality assurance effort starts to control trhe tasks. When larger number of tasks is being controlled and quality is assured, the detected error density decreases. With a decrease in error density, the level of bad fixes drop as well. The reason for this positive relationship is the nature of error types and motivational factors. When error density is high, the multitude of errors provides several obvious coding errors. These errors are relatively easy to fix, but are none the less time-consuming. When a programmer is stuck with reworking obvious errors he loses concentration over time. The effort in itself is low, and the monotony of the work makes it feel like a chore. Doing rework also removes the worker from the software production, which is the prioritized job for a software programmer. This leads to mistakes as attention to the work lowers, and boredom is present (Love, Edwards, Han and Goh, 2011: p. 177). When error density is low, the design errors provide a challenge, and the personal motivation to solve interesting problems allow for more concentration. High density also provides more reworks being conducted, and the chance for mistakes increase with the multitude.

With a decrease in bad fixes, the errors detected by the testing-team decreases as well. Decreased errors from bad fixes provide less detected errors waiting for rework. Fewer errors awaiting rework causes a drop in schedule pressure and management decrease the daily man-days for software development. The loop is closed as software developed is decreased by the adjustment.

R1: The first reinforcing loop concerns the detected error density and bad fixes in tandem with detection of errors in testing. When the error density increases, the level of bad fixes generated increases as well. The increase of bad fixes lead to an increase in the detected errors in testing. These errors are sent from testing and back to the rework pile, where they wait for the rework team to fix them. The increase in detected errors awaiting rework causes and increase in detected error density, and the loop is closed by this increase in. This loop indicates the managerial hazards of underestimating bad fixes in the rework process.

R2: The fifth and final reinforcing loop of this system concerns schedule pressures, bad fixes and detection of errors in testing. When the bad fix generation rate increases, this increase in bad fixes lead to a situation where the testing-effort finds more errors they have to send back to rework. With more detected errors awaiting rework, the schedule pressure increase. Increase in schedule pressures causes strain on the workforce. The added stress of being behind schedule prompts the rework effort to increase their speed. The stress leads to more bad fixes and the loop is closed. This loop also identify a managerial challenge, as bad fixes may lead to increased schedule pressure that in turn leads to more bad fixes.

These two enhancements is the sum of my changes in my applied hypothesis to the rework effort and process. By adding the logic and influence of detected error density on rework effort and applying more dynamics into bad fixes generation rate, the activity inside the rework process has been clearly identified. This provides added more realism to the model as a whole.

The stock and flow diagram below shows the relationship between the variables described in the CLD:

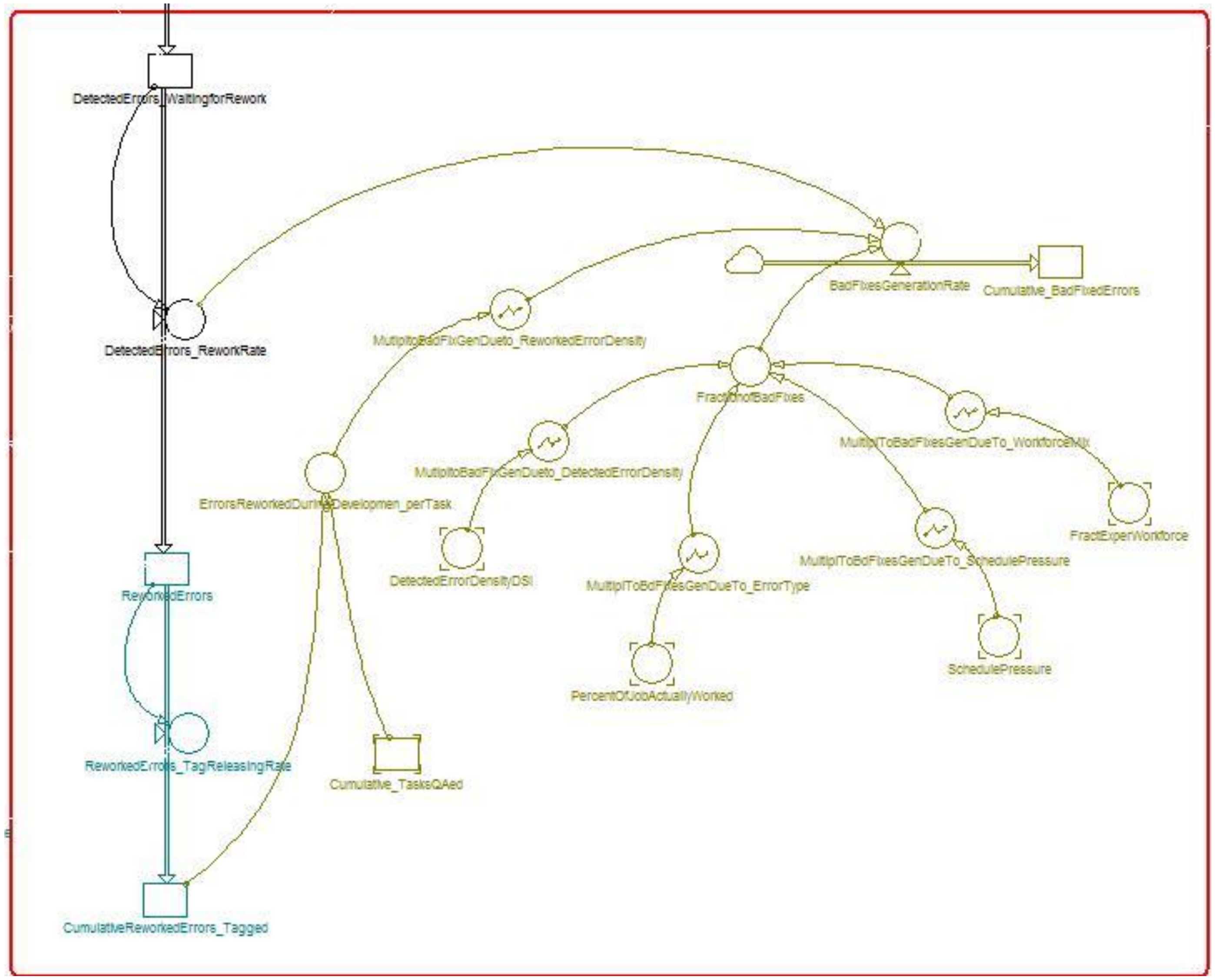

DetectedErrors_WaitingforRework
DetectedErrors_ReworkRate
BadFixesGenerationRate
Cumulative_BadFixedErrors
MultiplToBadFixGenDueto_ReworkedErrorDensity
FractionofBadFixes
MultiplToBadFixesGenDueTo_WorkforceMix
MultiplToBadFixGenDueto_DetectedErrorDensity
ErrorsReworkedDuringDevelopmen_perTask
FractExperWorkforce
DetectedErrorDensityDSI
MultiplToBdFixesGenDueTo_SchedulePressure
ReworkedErrors
MultiplToBdFixesGenDueTo_ErrorType
SchedulePressure
PercentOfJobActuallyWorked
ReworkedErrors_TagReleasingRate
Cumulative_TasksQAed
CumulativeReworkedErrors_Tagged

Figure 27. Stock and flow diagram is provided by enhancing the bad fixes generation rate

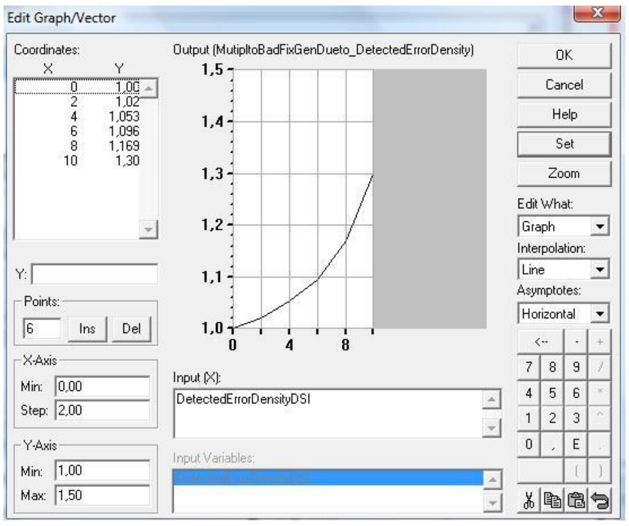
The original model's determination of bad fixes generation rate is under influences of rework rate and percent bad fixes through this formula:

*Bad Fixes Generation Rate = Rework Rate * Percent Bad Fixes*

I have now broken the "Percent Bad Fixes" constant in order to provide more realism to the generation of bad fixed errors.

The fraction of bad fixes is clearly based on four determinants for my enhanced model. The detected errors density, the percent of job actually worked, schedule pressures and fraction of experienced workforce. I will now explain the influence and the graph-function each multiplier holds to the fraction of bad fixes.

The density of errors influences the rate of bad fixes. When the error density is low, the few errors that quality assurance has identified are design errors. These errors are more tricky to rework, and demand ingenuity and focus from the programmer. When the detected errors density is high, the magnitude of errors is largely coding errors that are obvious, but time-consuming to fix. This creates a feeling of "maintenance", and the effort to correct them a chore. This is a negative motivational factor, and over time the staff will lose concentration and bad fixes is more likely to occur. The multitude of detected errors needing rework also provides the opportunity for more slip-ups as the workload is increased and the average number of bad fixes created increases. The graph for this multiplier is provided below:



Coordinates:

X	Y
0	1,00
2	1,02
4	1,053
6	1,096
8	1,169
10	1,30

Output (MutipltoBadFixGenDueto_DetectedErrorDensity)

Points: 6

X-Axis Min: 0,00 Step: 2,00

Y-Axis Min: 1,00 Max: 1,50

Input (X): DetectedErrorDensityDSI

Edit What: Graph

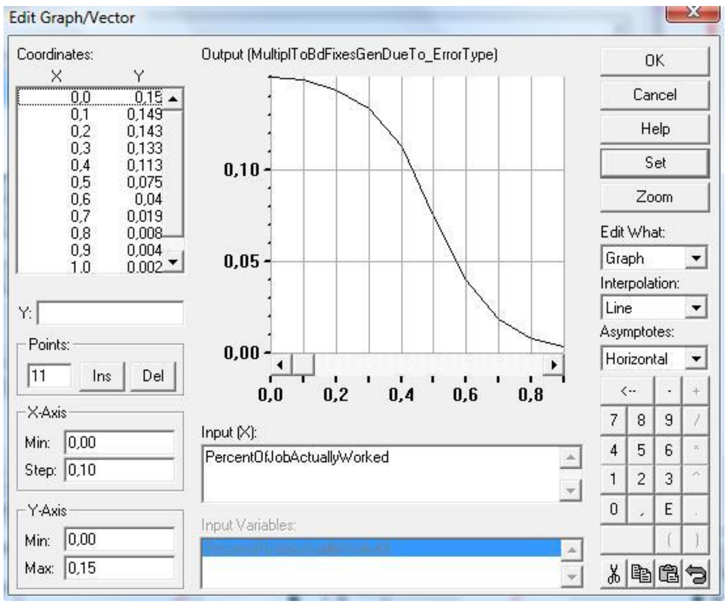
Interpolation: Line

Asymptotes: Horizontal

Figure 28. Multiplier to bad fixes generation due to detected errors density

When error density is low, the amount of bad fixes due to detected errors density is close to zero. When the detected errors density increase, the amount of bad fixes increase as well. The multiplier reaches a factor of 1.3 at the top. It is hard to put an exact measure of this multiplier due to the nature of bad-fix generation and motivational losses. From the literature we do learn however that when tasks are being performed under poor motivation, the quality suffers. It is not a dramatic effect that causes many errors to multiply, but rather a process where lack of concentration causes the occasional slip-up (Love et al 2011: p.117). A multiplier of 1.3 seems to be within the empirical description of the phenomenon, see Shull et al. (2002). They propose a multiplier between 1.2 and 1.5.

The second leg for determining the bad fix generation rate stems from type of the errors and the nature of design and coding errors themselves. The previous example was considering detected errors density and motivation from the programmer's side. The second leg looks at the relationship between design and coding errors and bad fixes. When the percent of job actually worked is low, the errors that are apparent in the system are design errors. These errors are harder to rework, and may cause bad fixes. From the examples of Abdel-Hamid, we saw that a bad fix can be contributed by faulty analysis of the error or by an initial fix of the original problem but a new error is created in its place (Abdel-Hamid, 1991: p.108). The nature of design errors as harder to detect, analyze and rework, make them more prone to bad fixes. This is depicted in my graph for the multiplier to bad fixes due to error types:



Output (MultiplToBdFixesGenDueTo_ErrorType)

Coordinates:

X	Y
0,0	0,15
0,1	0,149
0,2	0,143
0,3	0,133
0,4	0,113
0,5	0,075
0,6	0,04
0,7	0,019
0,8	0,008
0,9	0,004
1,0	0,002

Y:

Points: 11

X-Axis — Min: 0,00 Step: 0,10

Y-Axis — Min: 0,00 Max: 0,15

Input (X): PercentOfJobActuallyWorked

Input Variables:

Edit What: Graph

Interpolation: Line

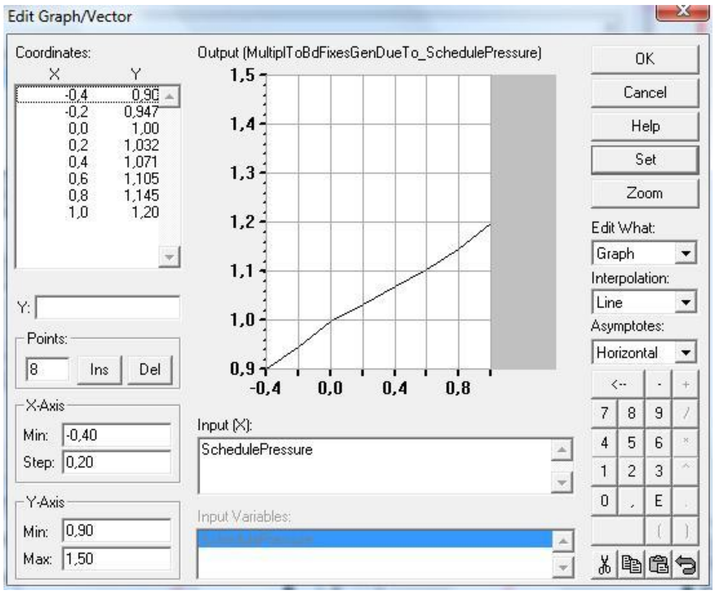
Asymptotes: Horizontal

Figure 29. Multiplier to bad fixes generation due to error type

From this graph, we see that design errors in the first half of production holds a multiplier to the likelihood of bad fixes at 0.15. This multiplier then decreases as the production of software moves from design to coding. In the end of the coding stage, it reaches a value of 0.002 and the multiplier holds a very small effect. This effect is much lower, due to the fact that coding errors are far simpler to correct. However, there is an effect on bad fixes due to the magnitude of coding errors. When there are more errors to correct, it's more likely to make a mistake. Design errors are particularly hard to fix, but their magnitude is less. This is linked to the description of error density and bad fixes.

The logic behind this multiplier follow the same logic as Abdel-Hamid applied for rework effort needed per error. Design errors are harder to rework due to their nature as blue-print errors (Abdel-Hamid, 1991: p.98). Design-flaws are errors in the software fabric that needs redesign. When we attempt to make corrections to the design, the likelihood of making a mistake increases. Hence we assume here that the bad fix generation due to error type is 0.15 at design phase. Coding errors on the other hand may be large in multitude, but are by themselves very simple tasks to correct. Therefore aside any external influence the multiplier should hold a low value. I assume in this case a value of 0.002 for coding errors.

The third leg for bad fixes is schedule pressures. When a project is behind schedule there are increasing pressures put on staff in order to finish the project on time. Such pressures cause stress for the manufacturing team and for the rework effort. When people are under stress, their ability to perform is gradually deteriorating (Jones, 2007). In a rework setting this translates into making erroneous analyses of errors and subsequently bad fixes. This is provided in my graph below that indicates how schedule pressures increase the likelihood of bad fixes.


Edit Graph/Vector

Coordinates:

X	Y
-0,4	0,90
-0,2	0,947
0,0	1,00
0,2	1,032
0,4	1,071
0,6	1,105
0,8	1,145
1,0	1,20

Output (MultiplToBdFixesGenDueTo_SchedulePressure)

Points: 8

X-Axis Min: -0,40 Step: 0,20

Y-Axis Min: 0,90 Max: 1,50

Input (X): SchedulePressure

Edit What: Graph

Interpolation: Line

Asymptotes: Horizontal

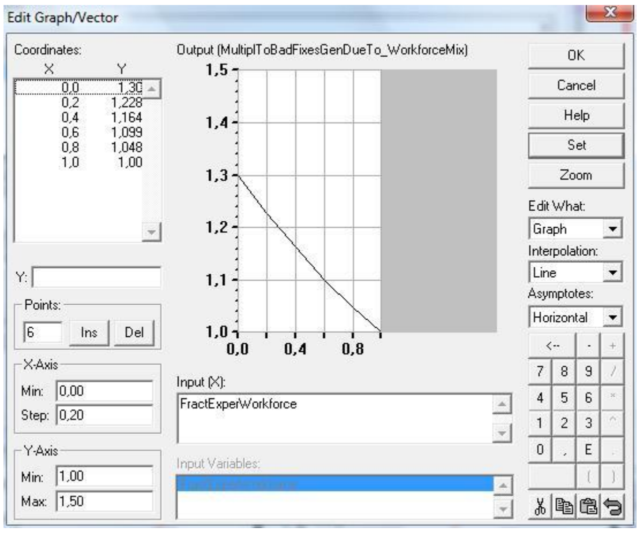
Figure 30. Multiplier to bad fixes generation due to schedule pressure

Schedule pressure increases the likelihood of bad fixes increases. With no pressures, the development team is unlikely to make any bad fixes as they have the time and clarity to perform their duty. However, when the pressure is at its peak, the chance of making a mistake increases to 1.2. Stress factors create pressures that lead to poor decision-making. From several different fields we find evidence of corner-cutting and quick fixes when schedule pressures are high. The effect of stress is cumulative, creating increasing pressures over time. When under stress, people usually persist in trying the same erroneous solutions to a problem. This leads to an additional cause of stress (Wickens and Hollands, 2000).

The multiplier for the schedule pressure holds a smooth increasing line to reflect this effect. The effect reaches its high-point at 1.2 when schedule pressures are at its peak. In this scenario, it is again hard to directly calculate the physiological effect of stress on bad fixes. From the empirical descriptions of the phenomenon, we learn that increased pressures do lead to bad decisions or corner-cutting. The worker is doing the tasks he is set to do, but the pressure of urgency provides occasional slip-ups as well. The industry average of bad fixes is 7% (Jones & Bonsignour, 2012), and there are always some degree of schedule pressure in a software development project. This indicates that the multiplier to schedule pressure and bad fixes should not be dramatically high. From the empirical data used by Abdel-Hamid in the base model, the averages for bad-fixes percentage ranged between 6.5% and 10% (Abdel-Hamid, 1991: p.108).

This average of bad fix percentage is the result of previous studies on successful projects. This average can be higher in individual projects as indicated by Jones and Bonsignour. In the base model the effect of schedule pressure on error generation rises from 1 under normal condition to 1.5 under stress situation. By assuming a smaller value for the effect on the percentage of bad fixes due to the empirical data, the value of 1.2 for stress situations on bad fixes can be reasonable. I argue that a top-point of 1.2 is within reasonable limits. With no schedule pressure, the multiplier takes on the value of 1, which implies no effect on bad fixes generation.

The final determinant of the fraction of bad fixes rest with the workforce mix. Experienced personnel have received training and expert knowledge over time in software development. They can rely on previous experiences when they are faced with errors. Therefore the mix of new and experienced personnel is affecting the rate of bad fixes. The graph below show the relationship:


Edit Graph/Vector

Coordinates:

X	Y
0,0	1,30
0,2	1,228
0,4	1,164
0,6	1,099
0,8	1,048
1,0	1,00

Output (MultiplToBadFixesGenDueTo_WorkforceMix)

Points: 6

X-Axis Min: 0,00 Step: 0,20

Y-Axis Min: 1,00 Max: 1,50

Input (X): FractExperWorkforce

Edit What: Graph

Interpolation: Line

Asymptotes: Horizontal

Figure 31. Multiplier to bad fixes generation due to experienced work force

When the workforce consists mainly of new personnel, the multiplier for a bad fix is 1.3. As the mix between experienced and new workers tip in favor of experienced workers, the proneness to make bad fixes decrease accordingly. If the staff consists of only experienced workers the multiplier holds no effect at a value of 1. The reason for the importance of workforce mix lies in experience. When a team of developers have been working together for a while, they build good working routines. Over time they learn how mistakes are created, and what fixes should be applied when they appear. This chain of learning is an indispensable source for avoiding bad fixes (Jones & Bonsignour, 2012).

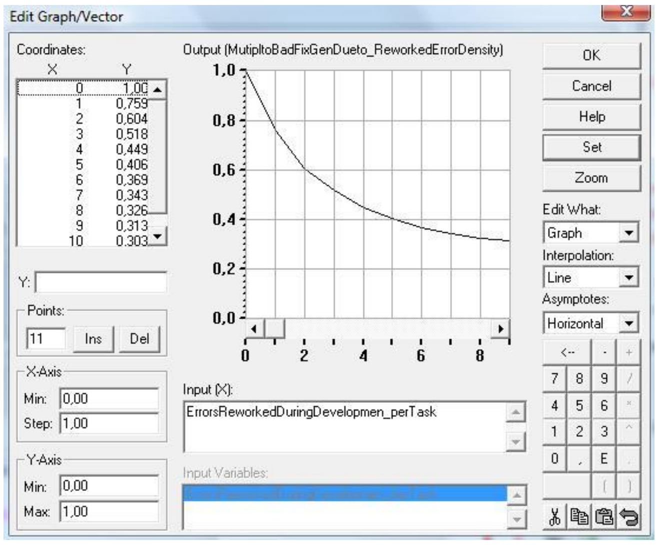
In order to arrive at the multiplier for workforce mix I utilized the data-material considering overall bad fixes. In the literature we saw that the average of bad-fixes in the industry as a whole is 7%. When Abdel-Hamid set his constant for percent bad-fixes, he used empirical data that provided a range of bad fixes between 6.5% and 10% (Abdel-Hamid, 1991: p.108). Bonsignour also indicated that the most successful projects make only 2% bad fixes while the completely failed projects as much as 20%.

I assume that the outliers here at 2% and 20% are due to other factors than work-force mix alone. The implementation of a test and quality assurance regime that is highly sophisticated, would aid a project down towards 2% bad fixes. In the same vibe, failure to establish any control-mechanism for bad fixes provides the 20% bad fix rate. Since we know the average is around 7% the upper and lower percentage due to work-force mix is constructed around this average. With a fully inexperienced staff, but with proper control functions they will commit 8.5% bad fixes. The experienced staff with a control system will make 6.5% bad fixes. This provides a multiplier of 1.3 between experienced and inexperienced. In addition, in the base model the effect of workforce mix on error generation rises from 1 when all workers are experienced to 2 when are all non experienced workers. I assume here a lesser value for this effect, according to percentage of bad fix generation. The average of 7% is much less than the average for error generation. This leads to a safer assumption for the value of 1.3 when all of the workers are experienced.

These four areas all determine the fraction of bad fixes. The variable is calculated as follows:

*Fraction of Bad Fixes = Multiplier to Bad Fixes Generation Due To Workforce Mix * Multiplier to Bad Fix Generation Due to Detected Error Density * Multiplier to Bad Fixes Generation Due To Error Type * Multiplier to Bad Fixes Generation Due To Schedule Pressure*

The bad fix generation rate is now determined by this new calculation to the variable rather than a fixed constant. Additionally, it is now determined by the rework rate and the impact of reworked and tagged errors density.



Output (MutipltoBadFixGenDueto_ReworkedErrorDensity)

Coordinates:

X	Y
0	1,00
1	0,759
2	0,604
3	0,518
4	0,449
5	0,406
6	0,369
7	0,343
8	0,326
9	0,313
10	0,303

Points: 11

X-Axis: Min: 0,00 Step: 1,00

Y-Axis: Min: 0,00 Max: 1,00

Input (X): ErrorsReworkedDuringDevelopmen_perTask

Edit What: Graph

Interpolation: Line

Asymptotes: Horizontal

Figure 32. Multiplier to bad fixes generation due to reworked and tagged errors density

The idea behind this multiplier is linked to the relationship to the capture-recapture procedure and rework. As errors are captured and identified by the quality assurance effort, rework is provided with information on the nature of the error. These errors are then reworked and sent back to the quality assurance team. The string of code containing the previous error along with its fix is then tagged and sent back into the system. If the testing effort finds applications to be deviating, and the deviation is caused by tagged errors it is sent back to rework due to the bad fix. In this manner the capture-recapture process aids the future rework over time as bad fixes or underlying design-flaws become apparent through the captured tagged errors.

The data from quality assurance and defect removal indicate than an increased review system, such as the capture-recapture procedure yields an 85% defect removal rate (Jones & Bonsignour, 2012). This is reflected in my multiplier. At the beginning of the project when there are a low amount of errors reworked and tagged, this factor holds no effect and is kept at the value of 1 .As more rework has been carried out and the additional review and tag commence, the available information on the true source of each error is revealed over time. This information helps the rework team to perform more proper correction, up to an 85% yield in the best cases in compared to previous the system. If we choose 70% as a general improvement in the rework effort through applying capture re-capture process, then the value of 0.30 for this multiplier can be reasonable. As we see from the graph, it has a smooth decreasing curve, which indicates better more efficient defect removal.

6.4 The quality assurance and testing processes

The next enhancements to my model, is the largest and most comprehensive contribution. From the literature we have learnt that defect removal is very important when it comes to software quality. From the examples of failures, we observe that errors and avoidable rework push projects out of schedules and plans. In order to achieve better software quality, it is therefore necessary to implement defect removal tools. Abdel-Hamid argued that there are two different methods regarding testing and quality assurance. From recent studies, e.g. Jones and Bonsignour, 2012, testing and quality assurance are interrelated processes. Quality assurance provides code reviews and small scale testing, while the testing effort focuses on testing finished applications. Combined, they provide management with valuable information on progress, milestones and defect removal.

In the original model, these two processes were seen as isolated efforts. This leads to a gap between the quality assurance and rework effort and system testing. The testing process is a dead-end in the base model, and no correction or feedback from that process is considered to the production process. Therefore, there is a need an enhanced realism, as no medium-range software project utilizes only one testing phase. In fact, the most common number of testing-efforts is six stages performed after certain predetermined milestones. This relationship will now be explained in more detail.

In modern software projects the concept of defect removal is central. The ability to remove errors during production and prevent errors, determine the factor of success or failure. In recent years, the leading companies in the software industry such as IBM ,AT&T, Microsoft and Hewlett-Pacard have developed standard routines for defect removal that combine both testing and quality assurance as complimentary procedures. In order to establish this relationship it is important to compare the two processes.

The quality assurance effort is described by Abdel-Hamid as a sequence of activities that are performed in conjunction with software development. This effort is carried out in order to assure that the software produced fulfill the specified standards. There are several techniques described by Abdel-Hamid, such as walk-throughs, reviews, inspections, code-reading and integration testing (Abdel-Hamid, 1991: p.71). These activities all require the quality assurance team and the production teams to have regular meetings where codes, functions and other sides of the software are discussed. This effort is an important tool for defect removal.

Jones and Bonsignour point out that having formal inspections of requirements, architecture, design and code alone provides minimum 65% defect removal (Jones & Bonsignour, 2012: p.249). In addition it is important to note that in modern quality assurance, parts of the testing-effort lies within the realm of quality assurance. Subroutine-testing that is carried out in 99% of all software projects are done by the quality assurance team (Jones & Bonsignour, 2012).

It is apparent that we can divide a software project into three layers, from micro to macro. The micro-level part of the software project is code-writing and how the different parts of codes are gathered in tasks. The quality assurance effort gains valuable knowledge on the micro-level as their effort is directly related to code and the fabric of the software. They perform subroutine testing which is the lowest level of integration testing providing an indication of functionality for each string of code. This is an important aspect to understand in order to establish the effort as a complimentary effort alongside the modern testing process. When formal inspection is carried out, the information gathered by the quality assurance team is also available for the testing teams. This provides the testing-teams with more complete and accurate information regarding requirements and specifications. This allows the testing effort to build better test-cases based on the pre-test efforts (Jones & Bonsignour, 2012: p.120).

The testing process is an important defect removal exercise. In modern software projects the testing effort is part in all projects as a tool for defect removal. During the last decades the methods of testing has changed. In the early days of the industry, testing was provided mainly as the last effort before a project was completed. One large testing-effort called Systems-Test would be carried out containing the complete product and the entire staff. This is also the approach in Abdel-Hamid’s base model, where the testing-stage is final effort. In the end of the project, all manpower is transferred to the systems test. The assumption in this model is that all errors undetected by the quality assurance team will be apparent in testing and fixed in this process.

Since 1991, the position of testing has changed. The software produced today are far more complex than the relatively simple software produced in the early days of the industry (Hjertø, 2003). Through years of experience, large companies such as IBM have devised testing-schemes that run continuously during the project. This has been highlighted earlier in my thesis as well, as the literature suggests a system of six testing-stages to be the average utilized amount for the testing effort. The increase in testing stages has two functions.

Firstly, the software under production has several applications and functions. It is very important to make sure that these applications are functional before the end systems-test. The final systems-test is the macro-level test of the complete software. The additional testing-efforts are the meso-level tests of the software. Completed tasks are gathered into functional pieces or applications of software. These applications are then tested in order to see if they function properly. Unlike the quality assurance process, the testing-effort is not focusing on individual code on the micro-level. The testing effort focuses on the overall function of an application, containing several strings of code (Jones & Bonsignour, 2012). The aim is to investigate the interaction and behavior of the codes together as an application. This is an important concept to note as it provides the clear relationship with the quality assurance effort. Quality assurance gains knowledge and controls the micro-level codes of the software. The testing effort investigates the applications on a meso-level, while the final systems test provides the macro-level analysis at the end of the project.

Secondly, testing of applications provide milestone. There has been a huge managerial problem in the software industry regarding the measure of progress. From the interviews conducted by Abdel-Hamid we learned that some projects measured progress in days expended as there was no other means of measure. With application-testing during the software project, as completed tasks forms an application real milestone and measures of progress are available. The number of applications worked and controlled in testing provides management with an early sign of the job size and the work still needed.

In Abdel-Hamid's model this concept is unfortunately lost as the project is only running one testing-effort. Secondly, the concept of running only one testing-effort push the project towards a system that looks more like a code-and-fix model. In the early days of software development, programs were in general much simpler than today. Ghezzi calls the model of production "The code-and-fix model" in his book "Fundamentals of Software Engineering" (Ghezzi et al. 2003). This approach implies that software is made in two stages. First stage is to write a piece of code, the second stage is to change that code to fix errors or add new functions. This process worked well with simple software programs, however as the industry grew in size the software grew in complexity.

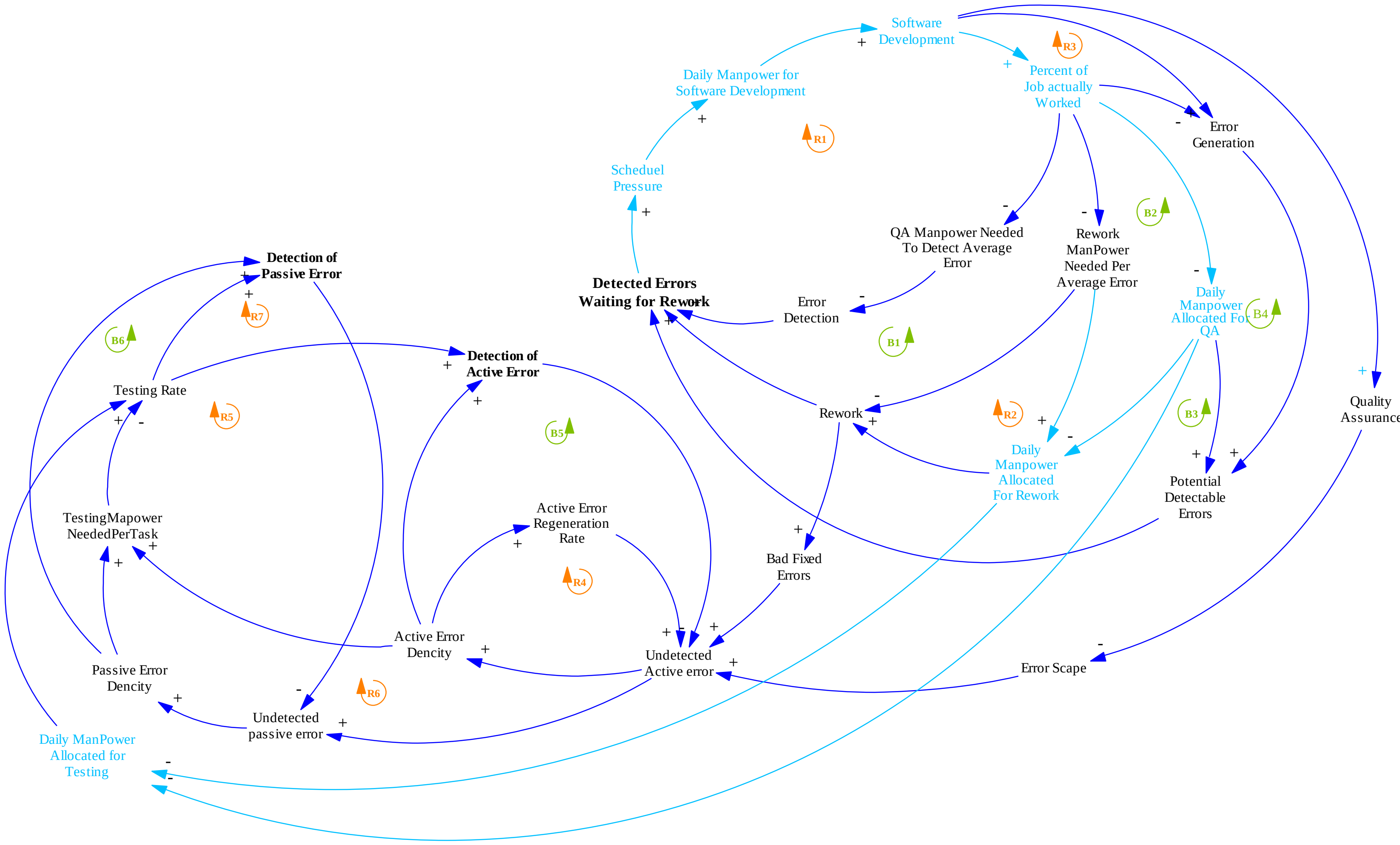
Unfortunately, the code-and-fix approach was able to endure through the first stages of computer evolution during the 80s and Hjertø argues that this fact is part of the software crisis. The industry adopted a simplistic approach to build more complex software and failures started to appear. When code becomes more complex, it is impossible to entangle all strings in order to write a code then rewrite it to fix an error. In modern software projects this approach is not applicable (Hjertø, 2003: p.296).

Hence, continuous testing and quality assurance is deployed to prevent failures and control codes as they are built. In Abdel-Hamid's model quality a piece of code is written, and then sent to quality assurance. In this effort the quality assurance team detects the error once. This error is then sent to rework and fixed. From this stage on, the task is sent to testing and will never be seen again by the rework team. This is a system that reminds of a code-and-fix approach. There is no feedback from testing to either quality assurance or rework. Given the facts above it becomes apparent that this gap needs to be fixed in order to provide a proper quality assurance and test-system. Testing and quality assurance are continuous efforts that complement each other in a modern software project. This part of the model is therefore in clear need of added realism to hold value in a modern software development environment.

In addition, by adding a capture-recapture structure to quality assurance, we reap the benefit of the different levels of knowledge from both quality assurance and testing. These two efforts are now helping management to measure the amount of errors in the software product and also how they originate. If they are untagged errors they are missed errors from quality assurance. If they are tagged errors, the error is originating as a result of a bad fix. This is a vital piece of information as manpower-allocation to the different efforts depends on solid information. Additionally, the information from tagged errors provides better knowledge and enhanced learning of how the individual pieces of code operate together in applications. The rework team is also able to understand how to fix the error more accurately.

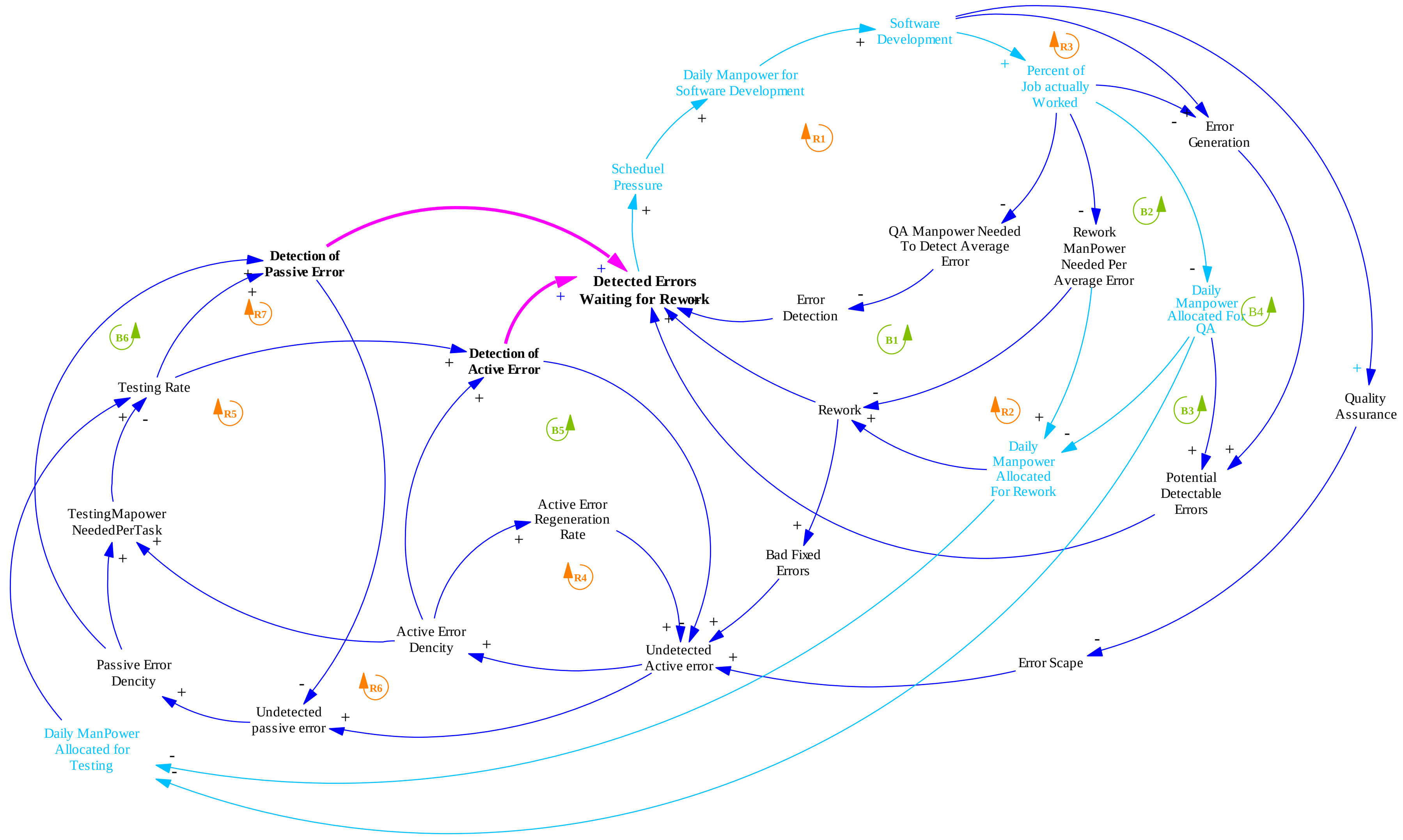
The CLDs provided below show the relationship between the variables between the testing and quality assurance effort. In order to highlight the gap between the efforts I have chosen to merge the two CLDs together. On the right hand side of the CLD we find the loops concerning the quality assurance effort. On the left hand side of the model we find the testing effort loops.

In the first presentation we observe how both bad fixes and escaped errors from quality assurance and rework lead to undetected active errors in the testing effort. The detected passive and active errors in testing are not fed back to the system since there is only one testing stage. With the previous explanation on modern systems testing, and the enhancement I provide the second CLD shows how these detected passive and active errors are indeed fed back to the rework-effort. The two purple arrows are now linking both variables to the rework-effort, closing the gap and providing the needed real-world relationship between the two processes.



Software Development
Daily Manpower for Software Development
Percent of Job actually Worked
Error Generation
Scheduel Pressure
QA Manpower Needed To Detect Average Error
Rework ManPower Needed Per Average Error
Detected Errors Waiting for Rework
Error Detection
Daily Manpower Allocated For QA
Quality Assurance
Detection of Passive Error
Detection of Active Error
Testing Rate
Rework
Daily Manpower Allocated For Rework
Potential Detectable Errors
TestingMapower NeededPerTask
Active Error Regeneration Rate
Bad Fixed Errors
Active Error Dencity
Undetected Active error
Passive Error Dencity
Undetected passive error
Error Scape
Daily ManPower Allocated for Testing
R1
R2
R3
R4
R5
R6
R7
B1
B2
B3
B4
B5
B6

Figure 33. QA and testing subsystems CLD,
With Gaps between testing process and QA and rework processes



Software Development
Daily Manpower for Software Development
Percent of Job actually Worked
Error Generation
Scheduel Pressure
Detection of Passive Error
Detected Errors Waiting for Rework
QA Manpower Needed To Detect Average Error
Rework ManPower Needed Per Average Error
Daily Manpower Allocated For QA
Error Detection
Testing Rate
Detection of Active Error
Rework
Quality Assurance
Daily Manpower Allocated For Rework
Potential Detectable Errors
TestingMapower NeededPerTask
Active Error Regeneration Rate
Bad Fixed Errors
Active Error Dencity
Undetected Active error
Error Scape
Passive Error Dencity
Undetected passive error
Daily ManPower Allocated for Testing
R1
R2
R3
R4
R5
R6
R7
B1
B2
B3
B4
B5
B6

Figure 34. QA and testing subsystems CLD,

With bridging gaps between testing process and quality assurance and rework processes

6.4.1 Manpower effort for testing process

The first enhancement is applied to the testing process and focus on the manpower effort for testing process. This manpower is needed to detect, classify and identify the average error during testing phase. Produced tasks are gathered and wait testing by the testing team. The testing team is members of production staff that are assigned to this process by management.

Testing process follows nearly the same logic as in detection and rework process. Therefore, design errors are harder to test and the average effort needed per such errors is greater than for coding errors. The error density also provides the same pattern of errors for testing as for rework and quality assurance. When there are several obvious errors found, the pattern among them may reveal a subtle design error causing the effect. Design errors are not passive, and they generate code-errors. Therefore the importance of error density in testing effort will be applied to Abdel-Hamid's model in order to enhance the realism and strengthen overall software product quality. The testing effort needed to test the average errors is also influenced as the quality assurance and rework efforts by the productivity of the staff. The staff productivity is measured as potential productivity minus the effect of motivational losses and communication overheads.

Testing manpower needed per average error is assumed as an exogenous constant in base model (Abdel-Hamid, 1991: p.115). But, according to the literature it is obvious that this cannot be a constant in reality and there are some dynamic effects that influence the manpower effort needed to test the average error. Therefore the needed personnel for the testing process (detecting, classifying and identifying the errors) hinge on certain key-features such as development productivity, density of detected errors in testing and error types. (Jones & Bonsignour, 2012) So I am going to modify it to an endogenous variable considering these key effects on it.

In the CLD below I show the entrance of testing manpower needed per average error based on the three factors mentioned above. It is also important to note that the two purple arrows (from detected errors in testing to detected errors waiting for rework) are used to close the gap between the testing and the quality assurance and rework processes. They indicate later enhancements in the testing structure. This will be explained in more detail under the enhanced system testing headline. They are necessary to include here in order to close the new loops. For now the relationship is indicated here by purple colored arrows in order to close the loop. The CLD below shows the new variables and loops due to applying the new enhancement in the large quality assurance and rework and testing subsystems CLD.

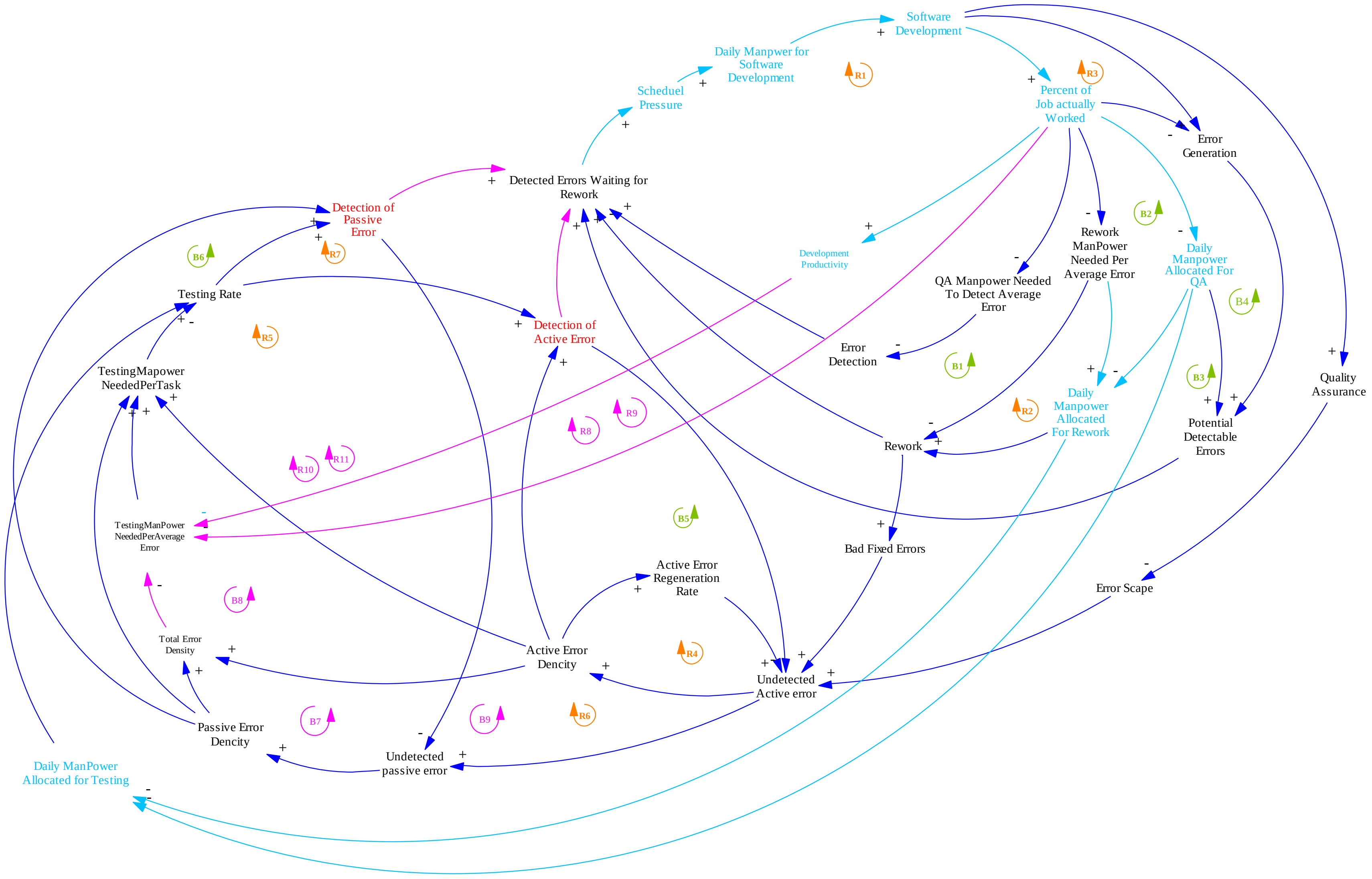

Software Development
Daily Manpwer for Software Development
Scheduel Pressure
Percent of Job actually Worked
Error Generation
Detected Errors Waiting for Rework
Detection of Passive Error
Development Productivity
Rework ManPower Needed Per Average Error
QA Manpower Needed To Detect Average Error
Daily Manpower Allocated For QA
Testing Rate
Detection of Active Error
Error Detection
TestingMapower NeededPerTask
Daily Manpower Allocated For Rework
Potential Detectable Errors
Quality Assurance
Rework
TestingManPower NeededPerAverage Error
Bad Fixed Errors
Active Error Regeneration Rate
Error Scape
Total Error Dencity
Active Error Dencity
Undetected Active error
Passive Error Dencity
Undetected passive error
Daily ManPower Allocated for Testing
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
B1
B2
B3
B4
B5
B6
B7
B8
B9

Figure 35 . QA and testing subsystems CLD,

with new loops are provided by enhancing the effort for testing process

From this CLD it is apparent that the testing manpower needed per average error is determined from the percent of job actually worked, the development productivity, and the density of detected errors in testing. The new relations by applying the enhancement are highlighted in purple. In order to establish these relationships more clearly, I will now present the zoomed in version of the CLD that captures the important variables and loops for these new relationships.

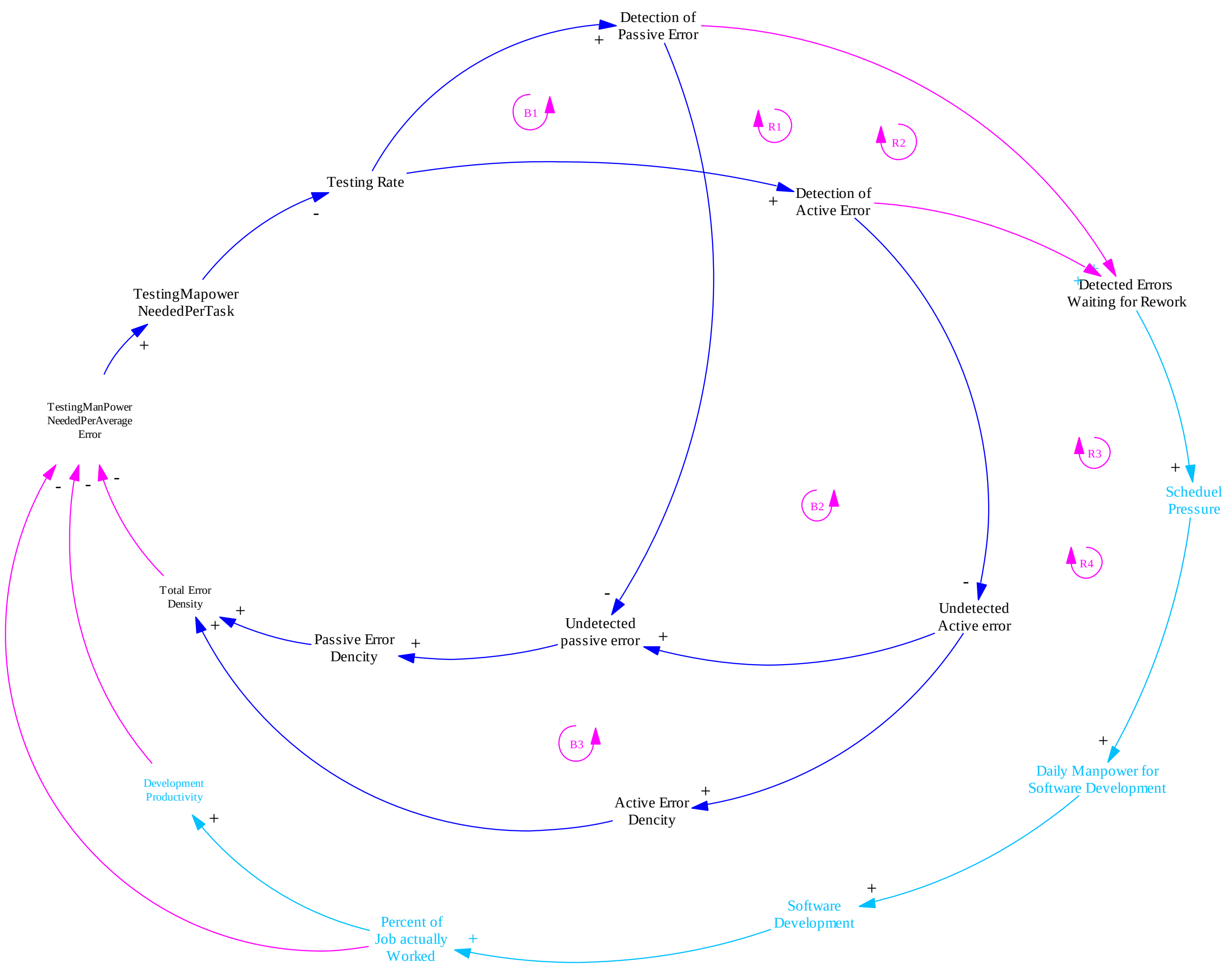

Detection of Passive Error
Testing Rate
Detection of Active Error
Detected Errors Waiting for Rework
TestingMapower NeededPerTask
TestingManPower NeededPerAverage Error
Scheduel Pressure
Total Error Density
Passive Error Dencity
Undetected passive error
Undetected Active error
Active Error Dencity
Development Productivity
Daily Manpower for Software Development
Software Development
Percent of Job actually Worked
B1
R1
R2
R3
R4
B2
B3

Figure 36. New loops are provided by enhancing the effort for testing process

The CLD above shows how the detection of passive errors and detection of active errors during testing phase influence the three important determinants the manpower effort is needed to test an average error. Some of the loops here represents the same logic, but are different in their consideration as they are linked to either active errors detection or passive errors detection in the testing phase. So in this CLD I will now trace the loops R1, R2, B2 and B3.

R1: The first reinforcing loop in this system concerns the detection of passive errors, and testing manpower needed per average error through the percent of job actually done and schedule pressure.

When the detection of passive errors in testing phase increases, the detected errors waiting for rework increases subsequently. The increase in errors waiting for rework causes schedule pressures, and management adjusts the daily manpower for software development upwards as a response to the increased schedule pressure. With an increase in the daily manpower allocated for software development, the rate of software development increases logically. This increase in software development increase the percent of job actually worked. When the percent of job actually worked increases, the project is moving from the design-phase towards the coding phase of development. Coding errors require less testing manpower per average error as they are easier to detect and identify. Less manpower needed per error translates to less testing manpower needed per task as well. This leads to an increase in the testing rate as the same amount of testing staff can test more tasks since they contain more coding errors. Higher testing rate provides more detection of passive errors during tasting phase, and the loop is closed. Loop R3 represents the same logic as in loop R1, but considers active errors detection in the testing phase rather than passive error detection.

R2: Loop R2 follows the exact same logic as loop R1, but deviates when the process arrives at the percent of job actually worked. When the percent of job actually worked increases, the development productivity increases as well. Percentage of job actually worked indicates that the project progresses and the staff is learning from this progression through completed tasks. It also indicates where in the development phase the project currently resides, in either the design or the coding phase of development. As we explained before, coding tasks are easier than design tasks to test. This leads to less manpower needed per average error, as the increase in productivity leads to more effective staff in the testing effort. Less testing manpower needed per error leads to less manpower needed per task as well. This increases the testing rate, and the loop closes as the increase in testing-rate leads to an increase in detection of passive errors during testing phase. Loop R4 represents the same logic as in loop R2 but considers active errors detection in the testing phase.

B2: The first balancing loop of the system concerns the detection of active errors in testing phase and the influence on testing manpower needed to test the average error due to the total error density.

When the amount of detected active errors increase, the amount of undetected active errors decrease. This leads to a subsequent decrease in the amount of undetected passive errors as well. Active errors have a retirement rate where they transform into passive errors. According to Abdel-Hamid, active errors have a retirement rate. The active error in question might cease to create additional errors after one or two generation of errors. For example will an error in a high-level module, produce interface errors at a lower level without leading to additional errors. Hence, the active error is now passive (Abdel-Hamid, 1991: p.113). Increased detection of active errors leads to a decrease in both undetected passive and active errors as a result. When the amount of undetected passive errors decreases, the passive error density decreases as well. This leads to a situation where the testing manpower needed to test the average error increases and the testing manpower needed per task follows suit. This leads to a decrease in the testing rate, and the loop is closed as the decreased testing leads to decreased detection of active errors during testing phase.

B3: Loop B3 follows the same logic as loop B2, but focus on the error density from detected active errors in testing phase. When the detection of active errors increases, the amount of undetected active errors decreases. As a result the active error density decreases with higher detection of active errors. The decrease in active error density leads to a decrease in the total error density. When error density is low, the project is in the design phase of development. This leads to a majority of design errors in the total error density that requires more manpower to test the average error. More manpower per average error translates into more testing manpower needed per task. This leads to a decrease in the testing rate and the loop is closed as less testing provides less detected active errors during testing phase. Loop B1 represents the same logic as in loop B3 but considers passive errors detection in the testing phase.

Thus, the average manpower needed to test the average error hinges on the density of detected errors in testing phase, development productivity and error type. The stock and flow diagram below shows how the testing manpower needed per task is decided by these factors. I will now present the variables in detail from this stock and flow diagram:

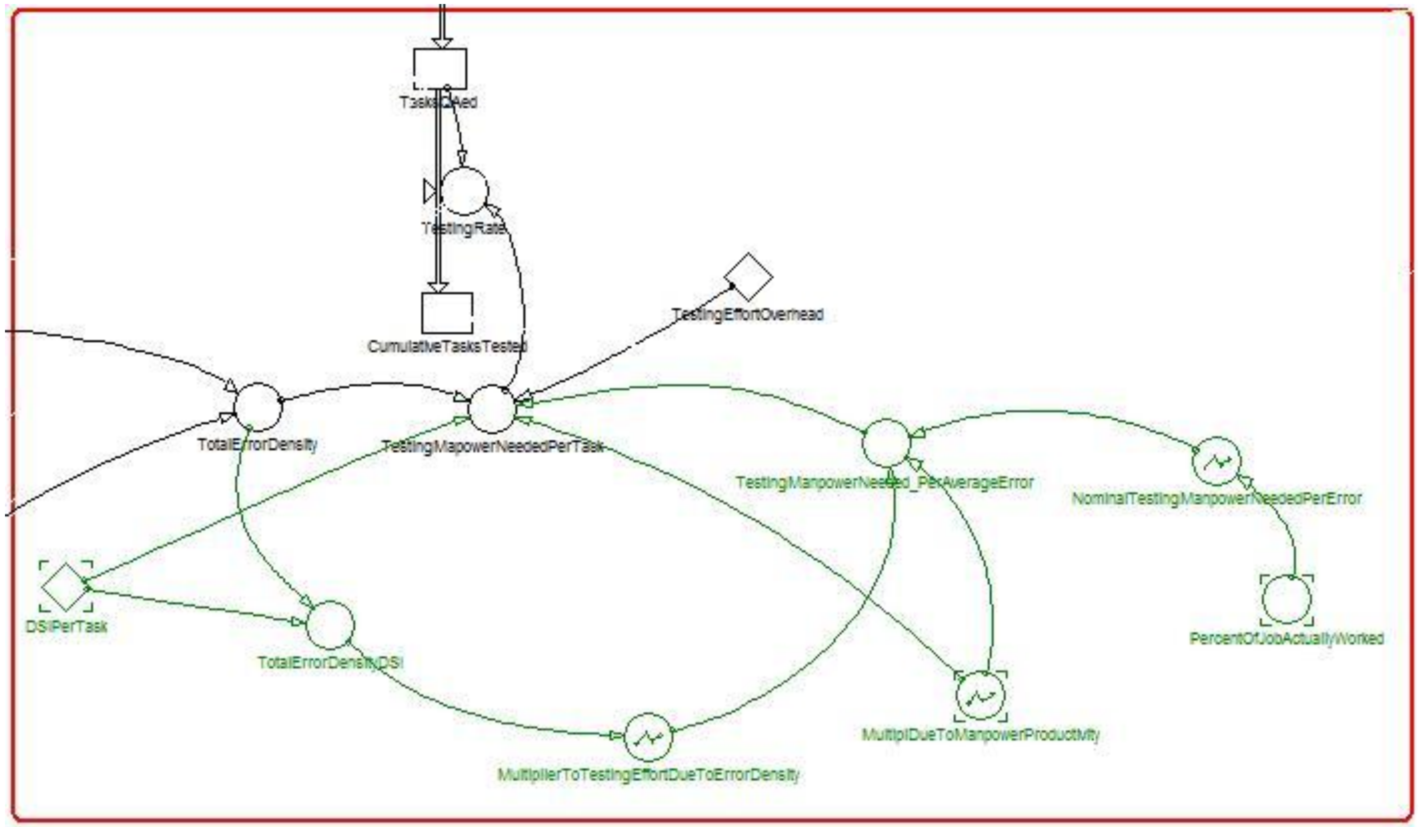
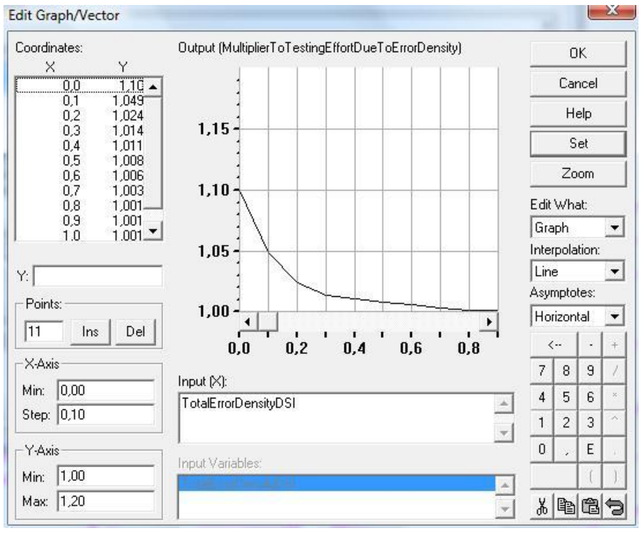

TasksQAed
TestingRate
CumulativeTasksTested
TestingEffortOverhead
TotalErrorDensity
TestingMapowerNeededPerTask
TestingManpowerNeeded_PerAverageError
NominalTestingManpowerNeededPerError
DSIPerTask
TotalErrorDensityDSI
PercentOfJobActuallyWorked
MultiplDueToManpowerProductivity
MultiplierToTestingEffortDueToErrorDensity

Figure 37. Stock and flow diagram is provided by enhancing the effort for testing process

This new addition to the model enhances realism significantly. In the original model a constant variable determined testing manpower needed per error. I am now representing a dynamic relationship to this constant, and expand our understanding of the determination of manpower needed to test an average error.

The first determinant for the testing manpower needed to test the average error is density of the detected errors in testing. The total error density in testing phase is decided by the amount of passive (coding) and active (design) errors that passed through undetected in the quality assurance and rework effort and they will be detected during the testing phase. The total errors density is per task, so it will be in error per KDSI, by dividing the variable with the constant DSI per task. DSI stands for Delivered Source Instructions and is used by the industry as a way to measure the size of a task. In some of the central software models for Abdel-Hamid's system dynamic approach the DSI method is applied as a size measure. One example is Boehm's COCOMO model, where Boehm defined the terms of DSI. "Delivered" as a term is generally meant: to exclude non-delivered support software such as test drivers. "Source Instructions" includes all program instructions created by project personnel and processed into machine code by some combination of preprocessors, compilers or assemblers (Abdel-Hamid, 1991: p.82).

KDSI then translates to 1000 Delivered Source Instructions and is used as the measurement-unit for the entire project as a whole (Abdel-Hamid, 1991: p.98-99). This density is transformed into a graph-function as the multiplier to the testing effort due to the density of detected errors in testing. The graph is pictured below:



Coordinates:

X	Y
0,0	1,10
0,1	1,049
0,2	1,024
0,3	1,014
0,4	1,011
0,5	1,008
0,6	1,006
0,7	1,003
0,8	1,001
0,9	1,001
1,0	1,001

Output (MultiplierToTestingEffortDueToErrorDensity)

Points: 11

X-Axis Min: 0,00 Step: 0,10

Y-Axis Min: 1,00 Max: 1,20

Input (X): TotalErrorDensityDSI

Edit What: Graph

Interpolation: Line

Asymptotes: Horizontal

Figure 38 . Multiplier to testing effort due to density of detected errors in testing

When the error density is low, there are mainly designs errors present in the tasks tested by the testing crew. From the original model we learned that design errors create cascade-effects of coding errors or software malfunctions (Abdel-Hamid, 1991: p.105). This problem is also transferred to the testing effort. With design errors percent the effort needed to detect, identify, classify and report an error increases.

In this enhanced model I have set a rate of 1.10 for the multiplier in increase for the effort when error density is low. When error density is high, the coding errors are easier to detect, classify, and design errors that hide behind several coding errors are easier to reveal. Hence the multiplier lowers to reach 1.0 and hold no effect at high error density-levels. The multiplier is then used as one of the determinants for the testing manpower needed to test the average error. The value of 1.10 as a multiplier for the low error density derives from the nature of passive and active errors. When design errors are transferred to testing, they are always active errors. Hence, the effort to detect and identify the error is slightly higher than for passive coding errors. The active design errors will cause a string of subsequent errors that by themselves look uncorrelated. To identify this correlation acquires some additional effort (Abdel-Hamid, 1991 p.113-114). Since the effort to test and detect the design active errors is slightly higher due to this challenge, the value of 1.10 is applied. I derive at this value based on the empirical data of Jones and Bonsignour. The data shows that the difference in execution time for testing based on types of test applied at different stages between design and coding. The minutes spent on the execution for the different test cases do not show huge variations, even though individual cases can provide a time-frame far beyond minutes depending on the error found and project size. I utilize this data even though it does apply some generalizations. Thus the effort for design errors are lightly increased due to the factors explained above. The logic follows the same pattern as for the rework effort needed per error. Since we know the effort to testing cases are smaller than to rework by the same data, we have a goal-seeking declining curve between 1.10 and 1.0 (Jones & Bonsignour, 2012: p.338-340)

The second determinant for the testing manpower needed to test the average error is development productivity. The variable development productivity is described as the psychological model of group productivity. Losses due to faulty process refer basically to communication and motivation losses. Potential productivity is the maximum productivity that can be obtained if there are no faulty processes. Aside from taking into consideration the losses to communication, the positive boosts to productivity stems from the workforce mix and the ability to learn by the progress in the project, and percent of job actually is done. Experienced workers are in general more productive than new workers, and their presence is a positive boost to overall productivity. The ability to learn is also an important factor. When the software has been developed, the development staffs gain knowledge and insight into the inner workings of the software. Through design and coding they obtain the ability to increase their understanding, and also identify errors or fixes. These factors are all calculated into the development productivity. Testing personnel are assigned from development teams, and their ability to perform the task is influenced by the same factor as when they belonged to production. (Abdel-Hamid, 1991) (Jones & Bonsignour, 2012).

The third and final determinant for the testing manpower needed per average error stems from the percent of job actually worked and error type. The average testing manpower needed per error differs by error type and between the design and coding phase and design errors are more costly to test. When the software has been designed and one move into the coding phases, the amounts of completed tasks for testing are increasing as well. When the design has been constructed, and coding is the primary activity, the effort to test each task decreases.

For each testing stage the testing staff is learning about the software's behavior as well. When additional components are finished and put together in integration-testing the overall knowledge and understanding of the software increases. The errors found can now be traced in a string from milestone to milestone making the effort needed per average error decrease between each testing stage (Jones & Bonsignour, 2012) .

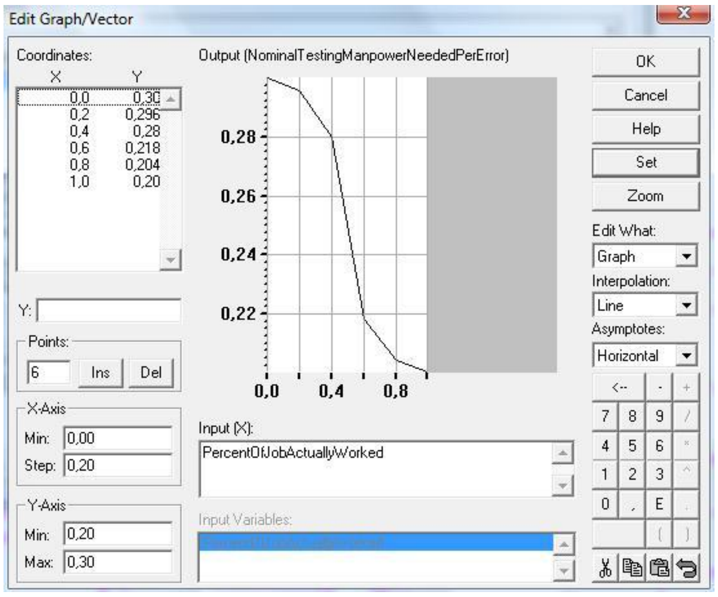
In the original base-model Abdel-Hamid determined a constant level for the testing manpower needed per error. Based on empirical examples from the DE-A project and other reported projects at the time, he set a constant level of 0.15 man-days/Error for the testing effort (Abdel-Hamid, 1991: p.115). The constant was chosen based on empirical results from Herdnon and Lane's article "Analysis of Software Errors for Cost Factors" and Shooman's "Software Engineering- Design Reliability and Management". However, since we are operating with an enhanced testing system this effort cannot be a constant and the effects of error type is important. The nominal manpower needed per error is influenced by the same factors as the rework effort, namely error type. The error type indicates where in the project we are currently situated. The progress from the design phase to coding phase is influencing the effort needed per error as well. There are two main reasons: (1) design errors are harder to test and identify as they are active errors causing multiple errors. (2) The coding errors are passive errors that are easier to test. Additionally progress provides learning for the testing-team as the continuous testing process provides knowledge per testing stage. Testing teams learn from one test to the other.

In order to calculate a multiplier I utilized the logic for the rework multiplier due to error type. In the original model this multiplier held a 0.6 for design, and 0.3 for code-errors. In this case however, the testing team is merely classifying the error not reworking it. In the empirical data provided by Jones and Bonsignour, the minutes spent on test case execution and the defect repair minutes are provided. In these tables it becomes clear that the minutes spent on execution of tests are far less than for defect repair. For example a subroutine case takes 1 minute per case to execute, while defect repair after the subroutine test takes 10 minutes.

In any type of test, ranging from general testing to advance testing, the time spent on execution is significantly lower (Jones & Bonsignour, 2012: p.338-340). Looking at the types of tests and the target error type, such as design or coding, the difference in effort between the two are not large. Hence the effort is assumed to be 0.3 for the design errors and 0.2 for coding errors. Coding errors are given the value of 0.2 since they are by default easier to classify but their magnitude is higher. Therefore the nominal testing man-day needed per average error in the design phase is 0.3 while at the end of coding holds the value 0.2.

Additionally, testing-crews learn from each testing stage to the next. This follows the same logic as for the rework effort. The teams learn from experience, at the same time as the types of error give less effort per error. The graph below follows therefore the same pattern as for Abdel-Hamid's rework multiplier (Abdel-Hamid, 1991: p.105). The difference here is the level, and that the curve is a more s-shaped steeper line between 0.3 and 0.2

Hence the effort is assumed to be 0.3 for the design errors. Coding errors are given a value of 0.2 since they are by default easier to classify but their magnitude is high. Therefore the nominal testing man-day needed per average error in the design phase is 0.3 while at the end of coding holds the value 0.2. The graph is provided below:



Output (NominalTestingManpowerNeededPerError)

Coordinates:

X	Y
0,0	0,30
0,2	0,296
0,4	0,28
0,6	0,218
0,8	0,204
1,0	0,20

Points: 6

X-Axis Min: 0,00 Step: 0,20

Y-Axis Min: 0,20 Max: 0,30

Edit What: Graph

Interpolation: Line

Asymptotes: Horizontal

Input (X): PercentOfJobActuallyWorked

Figure 39. Nominal testing effort due to error type

The graph indicates an increase in effort at 0.30 in the design phase, and drops to 0.20 in the coding phase.

These three determinants provide the manpower needed to test the average error. The variable is calculated as follows:

*Testing Manpower Needed Per Average Error = Nominal Testing Manpower Needed Per Error * Multiplier to Testing Effort Due To Error Density * (1 / Multiplier Due To Manpower Productivity)*

From this calculation the new variable, instead of previous constant, influence the testing manpower needed per task. This variable is calculated by two constants, (DSI per task and Testing effort overhead) and two continuous variables (development productivity and testing manpower needed per average error). The first constant here is the DSI per task. In software development a "task" is a unit to size up a software product. A task can hold any arbitrary unit and number. Therefore it is necessary to provide a fixed number and unit for each task (Abdel-Hamid, 1991: p.82). In this case the unit chosen is DSI, as explained earlier. From the DE-A project and other empirical data on medium software projects the value is set to 60. For every task we have 60 DSI.

The second constant is the testing overhead. In the base model, the nominal testing overhead is a fixed component. It is calculated as a function between nominal-man-days/KDSI. Abdel-Hamid utilizes Boehm's estimation here as an assumption. Boehm estimates that the overhead effort is about 2-man-days/KDSI. For example in a 32 KDSI project, the overhead would amount to 64.41 man-days. If we assume that motivation and communication losses will result in a 50% loss in productivity, we have a documented overhead of 2 man-days/KDSI transformed into 1man-day/KDSI (Abdel-Hamid, 1991: p.115).

The variable is formulated as follows:

*Testing Manpower Needed Per Task = (Testing Effort Overhead * (DSI per Task / 1000) / Development Productivity) + (Testing Manpower Needed Per Average Error * Total Error Density)*

6.4.2 The capture-recapture process

In the original work of Abdel-Hamid, the quality assurance effort was given certain tasks to perform, such as walkthroughs, reviews and subroutine testing. Since the testing phase was considered to be a final system's test, the relationship between the efforts as defect removal tools was not accounted for. Recent development in this field indicates that both processes are interlinked. To establish this link, I will incorporate a capture-recapture system.

The idea behind this system comes from biology, where members of a given species are captured and radio-tagged. Later the research team recaptures the same number of specimens from the same area. Based on the total amount of tagged animals in the sample, it is possible to estimate the true size of the population. This example on how the total population may be measured in biological studies is directly transferrable to software development (Jones & Bonsignour, 2012: p.264).

In software development, errors are tagged in the same manner. This technique grew in popularity at the end of the 90s. There are two functions associated with error tagging. Firstly, the effort provides a gate-keeper function. When errors are reworked they will be tagged and released. Then they go through the production testing cycles. The testing process will test the complete tasks from development and filter out significant errors that need to be reworked. If these errors were previously tagged, they are a result of bad fixes and should be sent directly to rework to get proper correction. By doing this, the causes of the errors are known to management and further allocation to the rework effort follows (Petersson & Wohlin, 1999).

By tagging errors, we also address the legacy problem. This problem occurs when a piece of code that causes an error is kept because of its prior success in integration. For example: During the software production strings of codes are written to make an application work. Often programmers save or reuse pieces of codes for certain types of commands that functioned in earlier stages of the project, or from similar projects they have accomplished prior to the current project. Re-using previous functional code for similar commands and functions saves time when the software is built. Then we arrive at a situation where one of these codes has returned to rework due to an error. The programme is now stuck with a problem. Should the code that functioned for a similar application be re-written, or should he attempt to replicate the error with the code intact and apply a fix. The first choice is time-consuming, and it means changing a set of codes that worked before. The second choice saves time and leaves the source intact. Often the second choice is chosen due to the time-aspect and its previous success. Unfortunately, such decisions may save time for rework but causes huge error-problems later in testing. If that particular piece of code is causing the problem, it will continue to cause coding-errors through the rest of production as an active design-error. The time saved for rework is quickly gobbled up by the effort needed to find the source of the active error in testing.

Secondly, the tagging of errors works like estimating the true population of animals. When errors are tagged and released, they constitute the known errors. However, from the literature we have learnt that there are always unknown errors. It is of course impossible for quality assurance to detect all of them. Escaped errors will therefore slip through unidentified. When this stock of rework considerable errors increases, during the testing process and classification stage, these major errors will be identified through the identification stage. During the identification stage, the number of tagged errors can be employed to achieve an estimate of the true number of errors. In addition, a tag reveals where the error occurred. Untagged errors indicate a quality assurance problem, where significant errors escaped detection. All untagged errors are therefore transferred back to quality assurance for quality review and (further detection). In addition, management may now use these pieces of information to allocate resources more accurately between rework and quality assurance (Scott & Wohlin, 2008). The CLD below illustrates this process.

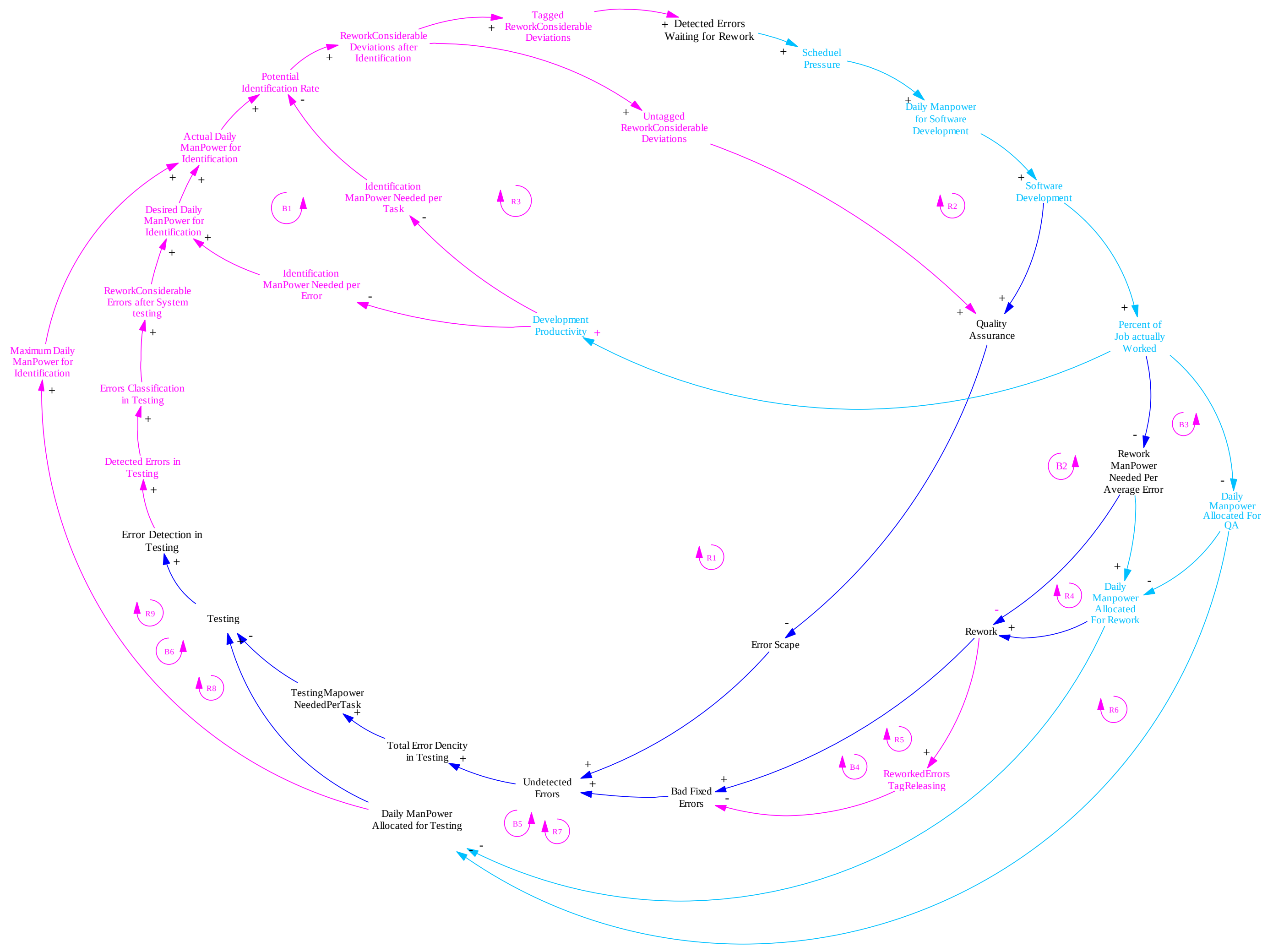

Tagged ReworkConsiderable Deviations
Detected Errors Waiting for Rework
ReworkConsiderable Deviations after Identification
Scheduel Pressure
Potential Identification Rate
Untagged ReworkConsiderable Deviations
Daily Manpower for Software Development
Actual Daily ManPower for Identification
Software Development
Identification ManPower Needed per Task
Desired Daily ManPower for Identification
Identification ManPower Needed per Error
ReworkConsiderable Errors after System testing
Development Productivity
Quality Assurance
Percent of Job actually Worked
Maximum Daily ManPower for Identification
Errors Classification in Testing
Detected Errors in Testing
Rework ManPower Needed Per Average Error
Daily Manpower Allocated For QA
Error Detection in Testing
Daily Manpower Allocated For Rework
Testing
Rework
Error Scape
TestingMapower NeededPerTask
Total Error Dencity in Testing
ReworkedErrors TagReleasing
Undetected Errors
Bad Fixed Errors
Daily ManPower Allocated for Testing
B1
R3
R2
B3
B2
R1
R4
R9
B6
R8
R6
R5
B4
B5
R7

Figure 40. New loops are provided by capture re-capture process

The CLD provided the relationship between the variables in this new system. By incorporating a capture-recapture procedure the relationship between quality assurance, rework and testing have changed. This CLD depicts how new loops are created, and how the capture-recapture concept influences manpower allocation, the quality assurance process, the testing phase and rework activities. I will here present six central loops, and explain their influence on the processes at hand. These are R1, R2, R3, R8 B2 and B4.

R1: The first reinforcing loop explained for this system concerns the relationship between Quality assurance, escaped errors, errors detected in testing and untagged rework considerable deviations.

By starting the project and developing the tasks, the quality assurance effort starts as well. By increasing the amount of the tasks developed, the level of tasks needed to be quality assured increases. This cause an increase in the effort to quality assure tasks. By more quality assurance effort the deviations that escape detection at this stage decrease. A decrease in errors escaped, leads to a subsequent reduction in undetected errors in testing. Less undetected errors for testing provide less total error density in testing, and as a result the testing manpower needed per task is less. Less manpower needed per task allows the testing-team to have a higher testing capacity. The same staff can cover more errors, and as a result the testing rate increases. Increased testing rate provides an increase in the error detection rate in testing. This leads to an increase in detected deviations in testing that needs to be classified as either rework considerable or no-rework considerable.

More detected deviations in testing leads to an increase in the rate of classification, and subsequently an increase in rework-considerable deviation after the system testing. These deviations must now be broken into their micro-components, and identified as either tagged or untagged deviations. More rework-considerable deviations in stock give management an incentive increase in the desired daily manpower for identification. This adjustment upwards in desired daily manpower, leads to more actual daily manpower for identification. More manpower for identification increases the rate of potential identification, and subsequently the rework-considerable deviations after identification. The reinforcing loop is closed as increase in untagged rework-considerable deviations after identification increases the quality assurance effort. This is due to the untagged deviations that are escaped deviations from previous quality assurance activities. They are sent back to this effort for further detection.

R2: The second reinforcing loop explained for this system follows the same logic as R1, but addresses tagged rework considerable deviations and the quality assurance effort.

Starting the project and developing the tasks provides an increase in software development. More tasks under production provide an increase in the quality assurance level. This causes an increase in the efforts to quality assure tasks. By more quality assurance effort, the deviations that escape detection at this stage decreases. This leads to less undetected errors for testing. This in turn reduces the total error density in testing, and less manpower is needed per task.

Less manpower needed per task increases the rate of testing, and more deviations are being detected in testing. The increase in deviations detection increases the stock of detected deviation in testing that need classification. This in turn leads to the increase in rework-considerable deviations after systems testing as the deviations are classified. More rework-considerable deviations increase the desired daily manpower for identification, and the actual manpower for identification as described in R1. More manpower increases the potential rate of identification and more deviations are identified. The tagged considerable deviations are the bad fixes from previous rework, and are subsequently sent back to rework. This increases the stock of detected errors waiting for rework.

When more errors are waiting for rework, management perceives that more effort is still needed before completion. This leads to schedule pressures that increases the daily manpower allocated for software development. This leads to an increase in software produced, and the reinforcing loop closes as more tasks produced provides more code for quality assurance to inspect.

B2: The first balancing loop explained for this system focus on the relationship between software development, the rework effort and bad fixes, errors detected in testing and tagged rework considerable deviations.

When software is being developed, the percent of job actually worked increases as a result. This signals that the project is moving from the design-phase to the coding-phase. This translates into a situation where the rework-manpower needed per error decreases as coding errors are easier to rework than design errors. Less manpower per average error provides an increase in the rate of rework activities.

However, when more rework is carried out, the chance of making a bad fix increases subsequently. This leads to more undetected errors until the testing effort, and the total error density in testing increases. The increase in error density leads to more man-power needed per task, and the rate of testing activities decreases as a result. Less testing provides less rate of deviations detection, and less detected deviations in testing. Less detected deviations leads to less classified deviations, and less rework-considerable deviations after the systems testing.

Management's desired daily manpower for identification drops as a result, and the actual daily manpower for identification is adjusted downwards. Less manpower equals less potential rate of identification, and less rework-considerable deviations are identified. This leads to less detected errors waiting for rework, less schedule pressures, and less daily manpower allocated for software development. The balancing loop closes as this reduction influences the software development rate, and causes it to drop.

R5: The third reinforcing loop explained for this system is important as it shows how the logic of loop B2 is changed from the entrance of the capture-recapture process to the rework effort and bad fixes.

Software is developed, and the percentage of job actually worked increases. This leads again to the decrease in rework-manpower needed per error, and the rework rate increases. When rework increases, more errors are fixed and tagged by quality assurance and released. More tagged errors provide better information about the nature and source of these errors. This information improves further rework efficiency. The result is a decrease in bad-fixed errors later. Less bad fixed errors leads to less undetected errors in the testing, and lowers the total error density for testing. This leads to a reduction in testing manpower needed per task and the rate of testing activities increases as a result. More testing provides more detected deviations, more deviations to be classified and more deviations classified as rework-considerable.

This increase indicates for management that more manpower is desirable for identification, and the actual daily manpower to this effort increases. So the potential rate of identification increases and the amount of rework-considerable deviations that are identified follows suit. This leads to more detected errors waiting for rework, more schedule pressures and an increase in the daily manpower for software development. The loop closes as the increase in manpower leads to an increase in software development. This loop shows the position of capture-recapture as a tool for avoiding bad fixes.

B4 The second balancing loop presented here portrays the relationship between software development, manpower allocation to rework, the capture-recapture process and the bad fixes, deviations detected in testing and tagged rework considerable deviations.

Software develops, and the percentage of job actually worked increases. When moving from design to coding, the rework effort needed per average error decreases. This gives an incentive to management for adjusting manpower allocated to rework downwards. Less daily manpower allocated for rework, less rework is carried out. Less rework provides less tagged and released fixed errors, and as a result more bad fixes are conducted later in the project.

More bad fixes provide again more undetected errors until testing phase, more total error density for testing, and a decreased in rate of testing activities. Less testing decreases the rate of deviation detection in testing. Less detection leads to less detected deviation in testing and less deviation for classification. Less classified deviation provide less rework-considerable deviation, and management's desired daily manpower for identification is decreased as a result.

Less desired manpower gives less actual daily manpower for identification, and the potential rate of identification activities drops. This leads to a subsequent decrease in rework-considerable deviation after identification, and less tagged rework considerable deviation sent back to rework. This decreases the detected errors waiting for rework, and schedule pressure is lowered. Less schedule pressure leads to less daily manpower for software development, and the balancing loop closes as the rate of software developed decreases at the end.

R8: The fourth reinforcing loop explained in this system concerns the software development rate, the manpower allocation for rework, testing and identification activities, software development and tagged rework considerable deviations.

When software is developed, the percent of job actually worked increases. This increase moves the project from design to coding, and the rework manpower needed per average error decreases. It leads to less daily manpower allocated for rework and results in an increase in the daily manpower that is available for testing. More daily manpower for testing increases the maximum daily manpower for identification.

More maximum daily manpower for identification allows management to increase the daily manpower for the identification effort. More manpower equals a bigger potential rate for identification activities and so the amounts of identified rework-considerable deviations increase. Therefore, more tagged rework considerable deviations are sent back to rework, and the amount of detected errors waiting for rework increases. More errors waiting rework leads to increase in schedule pressure, and management allocates more manpower to production. More daily manpower to production closes the loop as the rate of software development increases.

R3: The fifth and final reinforcing loop to be explained in this system concerns the production of software, development productivity, identification efforts and tagged rework considerable deviations.

More software being developed increases the percent of job actually done. Firstly, this presents that the project is progressing and moving from design phase to coding phase of development and the tasks that need to be accomplished are easier to do. Secondly, the production staff has learned through the production cycle and these factors increase development productivity. More development productivity leads to less identification manpower needed per task, and subsequently an increase in the potential rate of identification activities. This in turn leads to more rework-considerable deviations after identification and more tagged rework considerable deviations are sent back for rework. Hence, more amount of detected errors waiting for rework increases schedule pressure, and management adjust the daily manpower for production upwards. The loop is closed as the increase in daily manpower for production increases the rate of software production.

B1: The last and final balancing loop presented here, follows the same logic as R3, but includes management’s perception of desired manpower and how this balances the process. Software is again being produced, and the percent of job actually worked increases. As explained above, this leads to an increase in productivity due to learning and the nature of coding-tasks. More development productivity leads to less identification manpower needed per error. Less manpower needed per error gives management an opportunity to adjust the desired daily manpower for identification downwards. Less desired daily manpower for identification reduces the potential rate of identification activities. This leads to less rework considerable deviations after identification, and less detected errors waiting for rework. Less detected errors waiting for rework provides management with less effort remaining, so schedule pressure decreases. Less schedule pressure leads to a decrease in daily manpower for software development, and the balancing loop closes as less software is produced.

Now, I will present the stock and flow diagram for this system. Central features and mathematical formulations will be clarified.

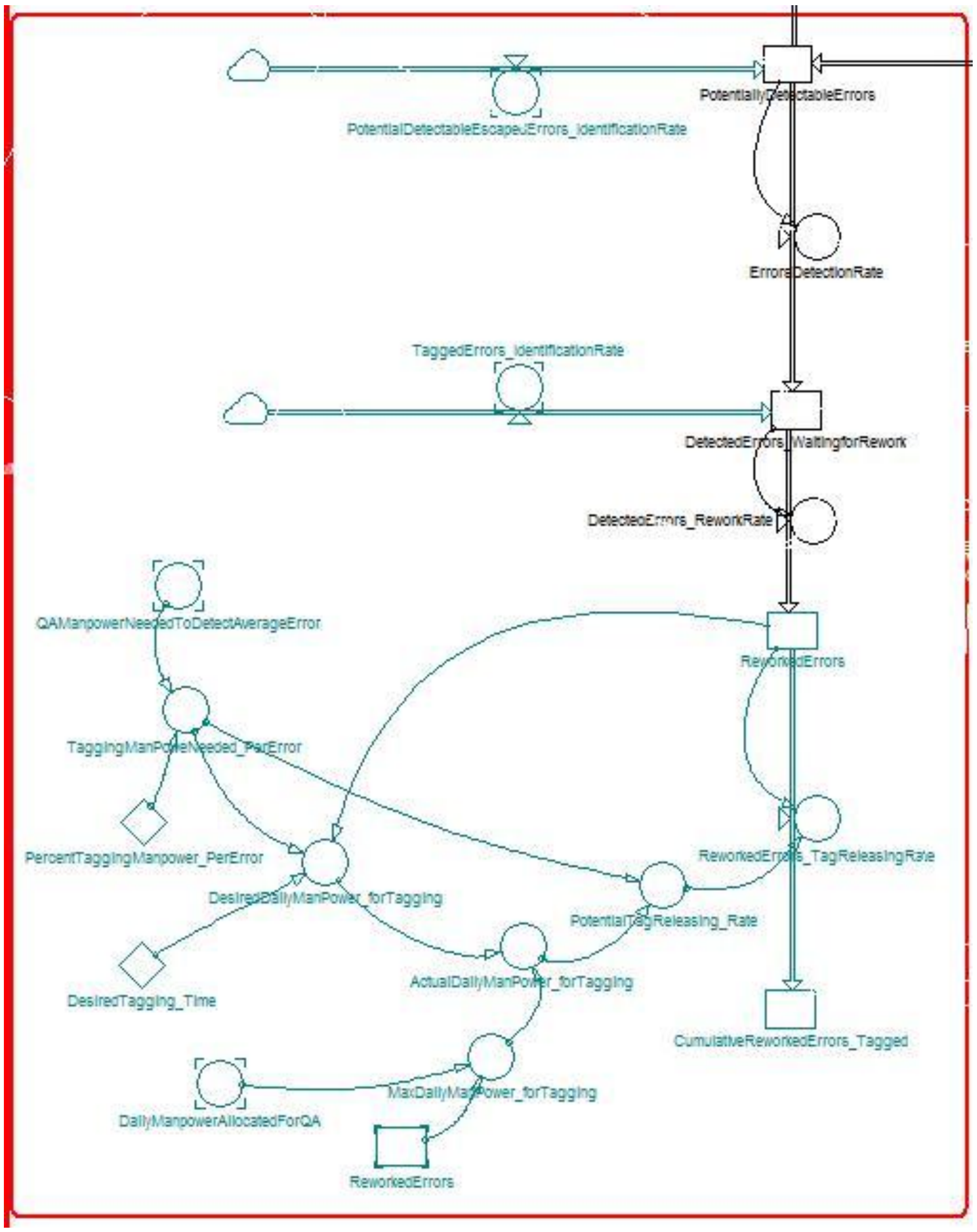

PotentiallyDetectableErrors
PotentialDetectableEscapedErrors_IdentificationRate
ErrorsDetectionRate
TaggedErrors_IdentificationRate
DetectedErrors_WaitingforRework
DetectedErrors_ReworkRate
QAManpowerNeededToDetectAverageError
ReworkedErrors
TaggingManPowerNeeded_PerError
PercentTaggingManpower_PerError
ReworkedErrors_TagReleasingRate
DesiredDailyManPower_forTagging
PotentialTagReleasing_Rate
DesiredTagging_Time
ActualDailyManPower_forTagging
CumulativeReworkedErrors_Tagged
MaxDailyManPower_forTagging
DailyManpowerAllocatedForQA
ReworkedErrors

Figure 41. Stock and flow diagram is provided by captures re-capture process, Part1

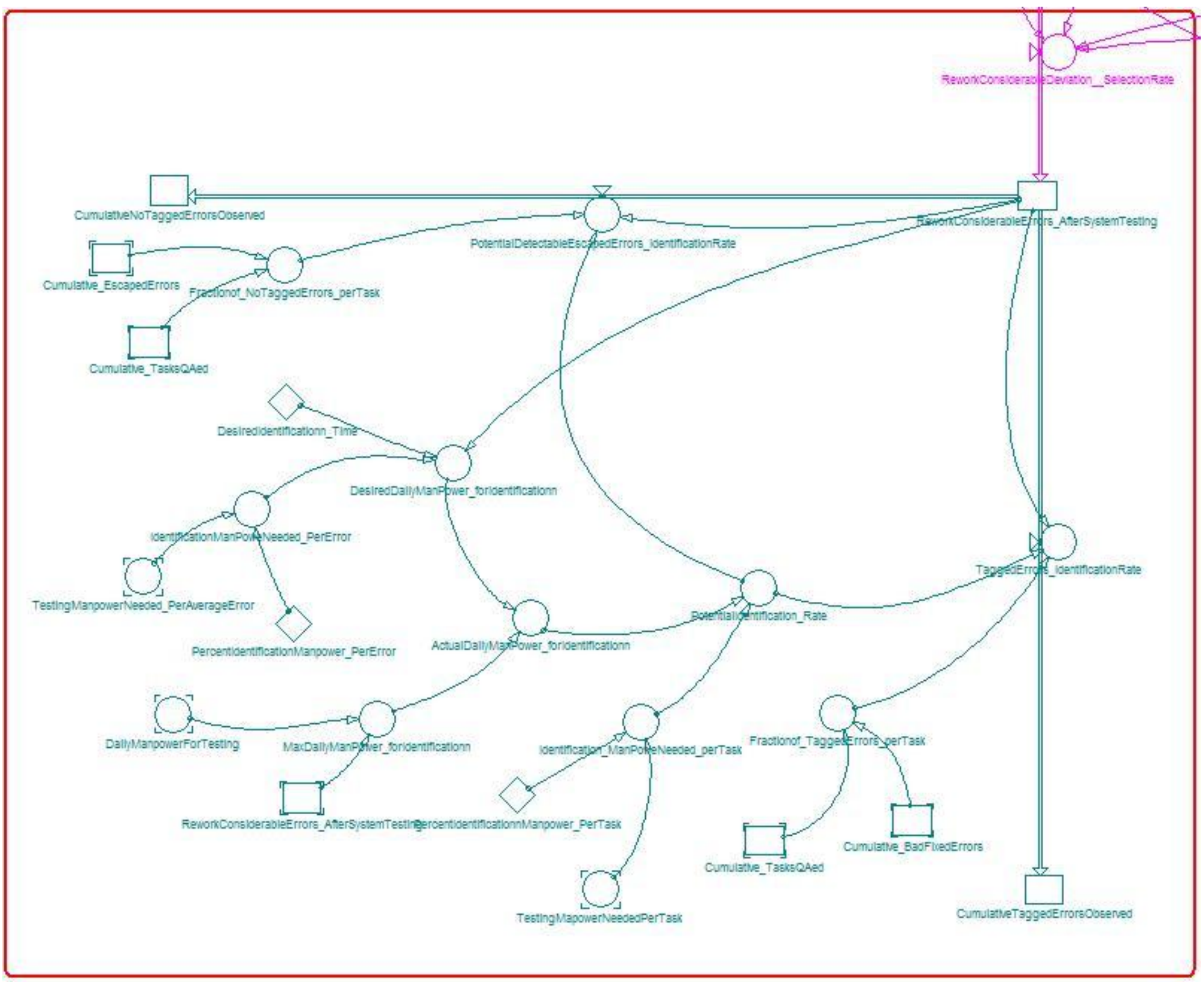

ReworkConsiderableDeviation__SelectionRate
CumulativeNoTaggedErrorsObserved
PotentialDetectableEscapedErrors_IdentificationRate
ReworkConsiderableErrors_AfterSystemTesting
Cumulative_EscapedErrors
Fractionof_NoTaggedErrors_perTask
Cumulative_TasksQAed
DesiredIdentificationn_Time
DesiredDailyManPower_forIdentificationn
IdentificationManPowerNeeded_PerError
TestingManpowerNeeded_PerAverageError
PercentIdentificationManpower_PerError
ActualDailyManPower_forIdentificationn
PotentialIdentification_Rate
TaggedErrors_IdentificationRate
DailyManpowerForTesting
MaxDailyManPower_forIdentificationn
Identification_ManPowerNeeded_perTask
Fractionof_TaggedErrors_perTask
ReworkConsiderableErrors_AfterSystemTesting
PercentIdentificationnManpower_PerTask
Cumulative_TasksQAed
Cumulative_BadFixedErrors
TestingMapowerNeededPerTask
CumulativeTaggedErrorsObserved

Figure 42. Stock and flow diagram is provided by captures re-capture process Part2

From the description above, as soon as software production starts to develop tasks, the quality assurance effort commences and the errors are detected. These are then sent to rework and fixed by the rework teams. The first stock of this capture re-capture system is called "Errors Reworked" that are waiting for tagging. After an error has been reworked, the task is sent back to quality assurance for tagging of the error and fix inside the task. The rate of which reworked errors are tagged is determined by the "Potential Tag Releasing Rate" and the "Reworked Errors" waiting for tagging.

Reworked Errors Tag Releasing Rate = MIN (Potential Tag Releasing Rate, Reworked Errors / TIMESTEP)

The "Potential tag releasing rate" is determined by the "Actual daily manpower allocated tagging" and "Tagging manpower needed per error".

Potential Tag Releasing Rate = Actual Daily Manpower for Tagging / Tagging Manpower Needed Per Error

The first determinant for "Potential tag-releasing rate" is the "Tagging manpower needed per error" is calculated by the "Quality assurance manpower needed to detect the average error" and the "Percentage of tagging manpower needed per error". As I explained during previous enhancements, the "Quality assurance manpower per error" is a function of error type and how effectively people work. Manpower is lost to communication and slack time. Error density is also important for this effort; the higher the error density, the easier it is to find and correct them. The effort needed to tag an error is much less than the effort is needed to quality assure an error. This is represented by the percentage of tagging manpower per error in this model. In my model this percentage is assumed to be 10%. Tagging manpower needed per error is therefore formulated in the following manner:

*Tagging Manpower Needed Per Error = QA Manpower Needed To Detect Average Error * Percent Tagging Manpower per Error*

The second determinant for the "Potential tag releasing rate" is the "Actual daily manpower for tagging". It depends on the "Maximum daily manpower for tagging" and the "Desired daily manpower for tagging". It takes the minimum value between these two variables value to control not allocating manpower to tag releasing stage more than the desired manpower is needed.

Actual Daily Manpower for Tagging = MIN (Max Daily Manpower for Tagging, Desired Daily Manpower for Tagging)

"Maximum daily manpower for tagging" is calculated as an IF-function between the daily manpower allocated for quality assurance and stock of reworked errors that are waiting for tagging. An if-function is applied here to control the allocation of manpower to tag releasing stage as soon as reworked errors are prepared to get tagged. The mathematical expression is as follows:

Max Daily Manpower for Tagging = IF (Reworked Errors > 0, (Daily Manpower Allocated For QA), 0)

"Desired daily manpower for tagging" is calculated by the "Tagging manpower needed per error" and the stock of "Errors reworked" and waiting for tagging and the desired tagging delay:

*Desired Daily Manpower for Tagging = (Errors Reworked During Development Waiting for Tagging * Tagging Manpower Needed Per Error) / Desired Tagging Delay.*

When errors have been tagged, they are transferred to the next stock of this system, namely the "Cumulative Reworked Errors Tagged". This constitutes the first step of the capture-recapture system.

The next part of this procedure is the recapture effort. When tasks are sent to testing, they are tested and then classified as either rework considerable or no rework considerable. After this selection through the classification stage in the testing process, the rework-considerable errors are sent to the first stock of this re-capture structure, named "Rework Considerable Errors after System Testing".

It is important to make a note here concerning the nature of testing and quality assurance processes. The quality assurance effort uses review, walk-throughs and the most simple test efforts, such as subroutine testing. These are targeted towards simple strings of code. The testing effort, on the other hand, performs more complex testing procedures, where the target subject is a working application constructed by several strings of code.

Therefore, when a piece of application has been tested, the recapture team must deconstruct the application through identification stage. When the task is deconstructed, the strings of code are examined for tags. There are two possible identities here. The first include errors that have no tag and must be sent to quality assurance. The second one includes tagged errors that will be sent to rework.

Errors without a tag are per definition escaped errors from the quality assurance effort. The outflow “Potential Detectable Escaped Errors Identification Rate” is governed by the “Fraction of No Tagged Errors per Task”, the “Potential Identification Rate” and the amount of “Rework Considerable Errors after System Testing” that waiting for identification.

*Potential Detectable Escaped Errors Identification Rate = (MIN ((Potential Identification Rate * Fraction of No Tagged Errors per Task), Rework Considerable Errors After System Testing / TIMESTEP))*

Since it is impossible to detect all errors i.e. errors that escape during quality assurance, they accumulate as escaped errors. The “Fraction of No Tagged Errors per Task” provides an average potential density of escaped errors per task that is calculated as a function between the “Cumulative escaped errors” during the quality assurance process and the “Cumulated tasks quality assured” after the development process. Hence the fraction is calculated as follows:

Fraction of No Tagged Errors per Task = MAX ((Cumulative Escaped Errors / (Cumulative Tasks QAed + 0, 0001))

The “Potential Identification Rate” is important because it governs the outflow rates in this system. It hinges on the “Actual Daily Manpower for Identification” and “Identification Manpower Needed per Task” and it is calculated as follows:

Potential Identification Rate = Actual Daily Manpower for Identification / Identification Manpower Needed per Task

As the formula shows “Potential Identification Rate” depends on effort per task that is provided by the variable named “Identification Manpower Needed per Task”. These variables provide the first half of the calculation for the potential identification rate. In order to calculate this, it is necessary to calculate how much manpower is needed to identify one task. This is done by utilizing the testing manpower needed per task, and the percentage of identification manpower needed per task. The testing manpower needed per task variable is a function of the number of errors in a task and represents the testing effort that would be expanded in the actual testing of average errors per task. The effort needed to identify an average task is logically much less than the effort needed to test an average task. This is reflected in the percentage of identification manpower per task in this model. In my model this percentage is assumed to be 10%. Identification manpower needed per task is thus calculated as:

*Identification Manpower Needed per Task = Testing Manpower Needed Per Task * Percent Identification Manpower Per Task.*

The “Actual Daily Manpower for Identification” is a minimum function between the desired and the maximum daily manpower for identification. These variables provide the first half of the calculation for the potential identification rate, namely the amount of manpower available for identification. This variable takes the minimum value between the desired and the maximum value of daily manpower. This is a control ensuring that manpower allocated to the identification stage does not exceed the desired manpower needed.

Actual Daily Manpower for Identification = MIN (Desired Daily Manpower for Identification, Max Daily Manpower for Identification).

The “Desired daily manpower for identification” is determined by the “Identification manpower needed per error”, “Rework Considerable Errors after System Testing” and the “Desired identification delay”. In my model this delay is assumed to be 3days. Desired daily manpower for identification provides the first part of the calculation for “Actual daily manpower for identification”.

*Desired Daily Manpower for Identification = (Rework Considerable Errors after System Testing * Identification Manpower Needed Per Error) / Desired Identification Delay*

The effort needed to identify an average error, is much less than the effort needed to test an average error. Hence, “Identification Manpower Needed per Error” is determined by the same logic as “Testing Manpower Needed per Average Error” under the effect of “Percent Identification Manpower per Error”. In my model this percentage is assumed to be 10%.This variable is calculated as follows:

*Identification Manpower Needed per Error = Testing Manpower Needed Per Average Error * Percent Identification Manpower per Error)*

The “Maximum daily manpower for identification” manpower rests on the “Daily manpower allocated for testing” and controlled by the stock of “Rework Considerable Errors after System Testing”. The If-function is applied here to control the allocation of manpower to identification stage as soon as rework considerable errors are prepared to get identified. This variable is calculated as follows:

Maximum daily manpower for identification = IF (Rework Considerable Errors after System Testing >0; (Daily Manpower for Testing); 0)

When no-tagged errors have been identified, they will be sent to quality assurance for further detection. They re-enter the quality assurance system through the inflow called “Potential Detectable Escaped Errors Identification Rate” for the stock called “Potentially detectable errors”.

The second scenario for rework considerable errors is that they are tagged errors. These tagged errors are the result of previous bad fixes and will be sent for rework. This is depicted by the flow of “Tagged Errors Identification Rate” that rests on the same manpower logic as the other outflow mentioned above and the “Fraction of Tagged Errors per Task”. The fraction of tagged errors per task provides an average potential density of bad fixed errors per task that is calculated as a function between the “Cumulative bad fixed errors” and the “cumulative tasks quality assured”. This follows the logic that tagged errors are caused by bad fixes:

Fraction of Tagged Errors per Task = MAX ((Cumulative Bad Fixed Errors / (Cumulative Tasks QAed + 0,0001)); 0)

Tagged errors identification rate is governed by the “Fraction of Tagged Errors per Task”, the “Potential Identification Rate” and the amount of “Rework Considerable Errors after System Testing” that waiting for identification.

*Tagged errors identification rate = (MIN ((Potential Identification Rate * Fraction of Tagged Errors per Task); Rework Considerable Errors after System Testing / TIMESTEP)).*

When the error has been identified as a tagged error, it is sent and reenters to rework through an inflow of “Tagged Errors Identification Rate” to the stock of detected errors waiting for rework.

6.4.3 The enhanced testing process

The last major enhancement I present is a new testing-regime, were testing is carried out in tandem with software construction. The literature concerning testing reveals that all software projects incorporate testing in their production cycles. Testing has been the central mode of defect removal since the 60s and 70s. However, the testing practices and procedures have been developed within the large manufacturing companies such as IBM, AT&T, ITT, without transparency to actors outside the industry (Jones & Bonsignour, 2012). These practices are now more open to research, as data over the years have been gathered and made public. From the book of Jones and Bonsignour we learn that from 600 of their client organizations, 100% utilized testing as a form of defect removal. The form and variety vary considerably however. The smallest projects include only one activity, while huge operations deploy as much as 16 discrete testing activities. In our case, we are dealing with medium ranged software projects, and the data suggest a scheme of 6 testing phases at certain milestones during production (Jones & Bonsignour, 2012: p.324).

In the introduction of the enhancements are applied in quality assurance and testing processes, I described the gap between testing and production in the base model. In the base model a piece of code would be produced, quality assured, reworked and then sent away to the final systems test that commences at the end of the project after production has halted. In a modern testing-environment this is not a feasible approach. It looks like a code-and-fix system that is not applicable to modern medium large software projects (Ghezzi et al., 2003). It is therefore important to bridge this gap in the model.

If I had kept the original blue-print here, I could have incorporated a separate testing-effort alongside the existing procedure, but it would have yielded little result. Without any connection between production and testing, the errors detected by the earlier testing stages would just have been compiled until the end. To provide a fully functional enhanced testing system, the errors detected must be sent back for quality assurance and rework, otherwise it would be completely unrealistic.

Therefore, in this thesis an enhanced system testing is applied in the based model considering the facts mentioned above. In order to establish this new testing-system, I will now present the overall CLD. The figure below contains the new structure (in purple) within the original software development system.

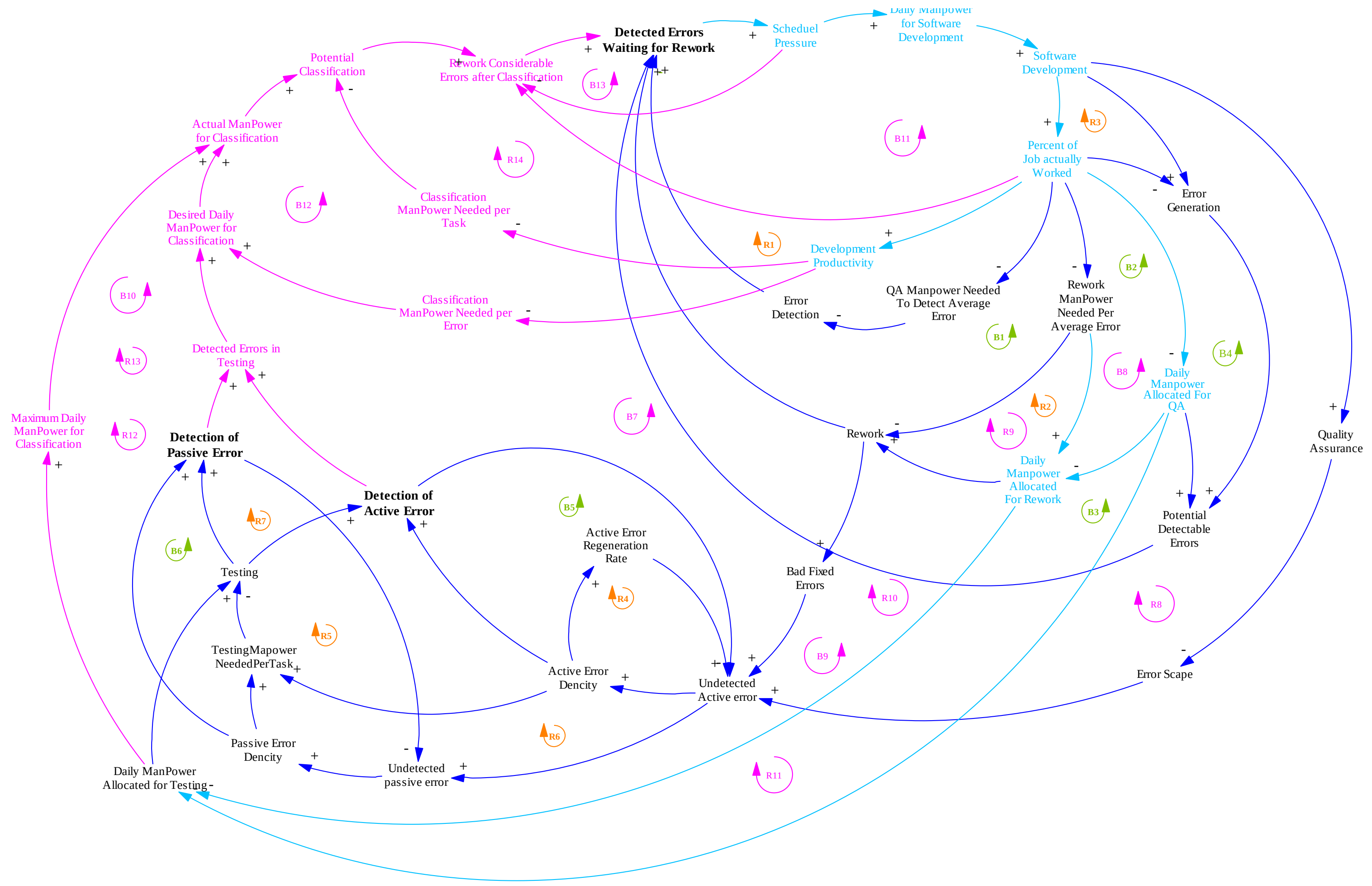
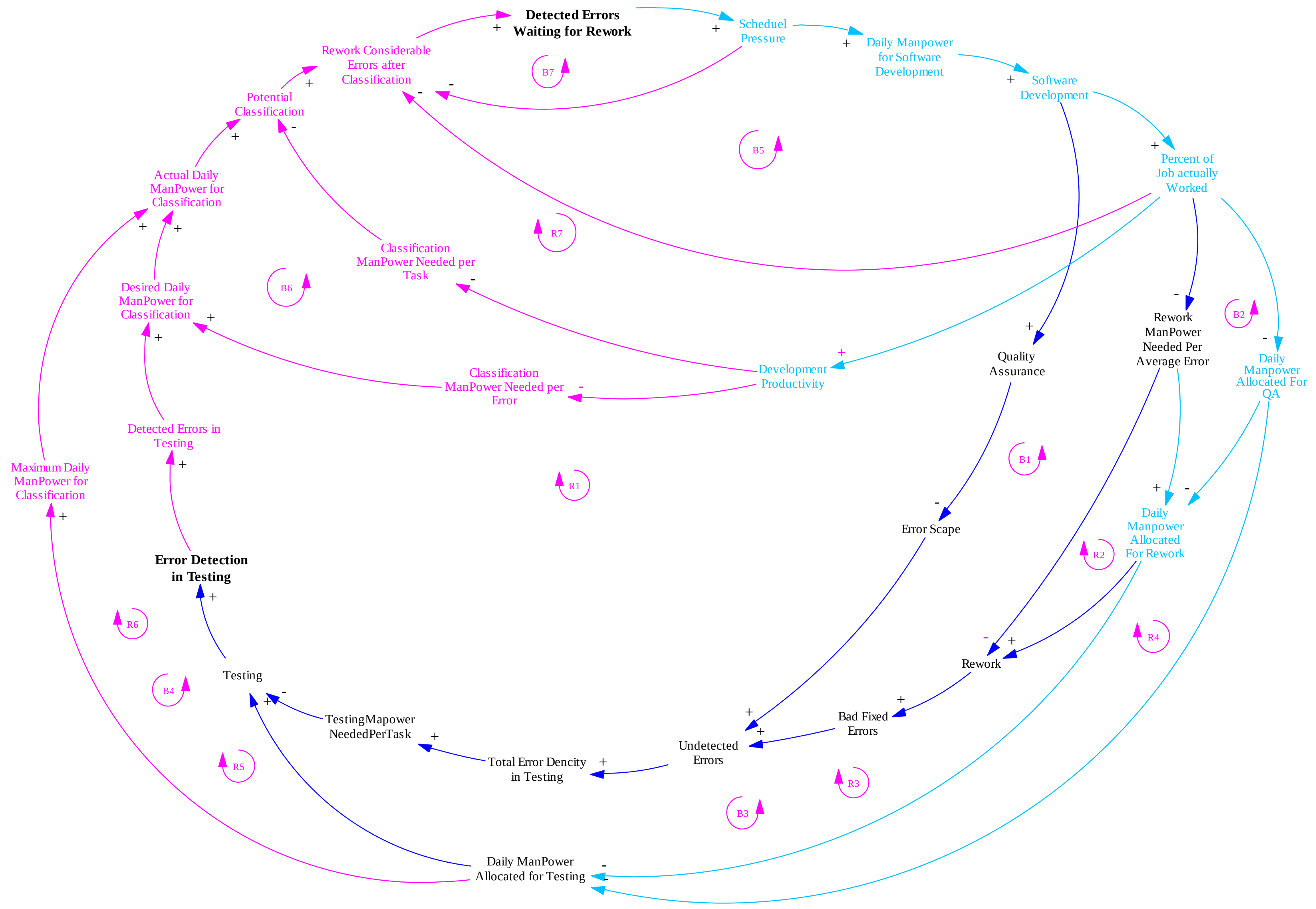

Detected Errors Waiting for Rework
Scheduel Pressure
Daily Manpower for Software Development
Software Development
Potential Classification
Rework Considerable Errors after Classification
Actual ManPower for Classification
Percent of Job actually Worked
Error Generation
Classification ManPower Needed per Task
Desired Daily ManPower for Classification
Development Productivity
Error Detection
QA Manpower Needed To Detect Average Error
Rework ManPower Needed Per Average Error
Classification ManPower Needed per Error
Detected Errors in Testing
Daily Manpower Allocated For QA
Maximum Daily ManPower for Classification
Detection of Passive Error
Rework
Quality Assurance
Daily Manpower Allocated For Rework
Detection of Active Error
Active Error Regeneration Rate
Potential Detectable Errors
Testing
Bad Fixed Errors
TestingMapower NeededPerTask
Active Error Dencity
Undetected Active error
Error Scape
Passive Error Dencity
Undetected passive error
Daily ManPower Allocated for Testing
B1
B2
B3
B4
B5
B6
B7
B8
B9
B10
B11
B12
B13
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14

Figure 43. Quality assurance and testing subsectors CLD, With new loops are provided by Enhanced testing process

From the figure, I provide the new variables that the enhanced testing process brings to the model. I have also highlighted three variables that in the original base-model where not connected, constituting the gap. Detected passive and active errors are not linked to detected errors waiting for rework with a new testing-regime that runs at different stages in parallel with production, detected errors in testing will be fed back to the production process through rework.

In order to further highlight the functions of testing in my model, I provide a CLD that focus on the new loops shown in purple. There are several new loops associated with the new system, and I will identify four important loops. The CLD below provides the new loops:



Detected Errors Waiting for Rework
Scheduel Pressure
Daily Manpower for Software Development
Software Development
Percent of Job actually Worked
Rework Considerable Errors after Classification
Potential Classification
Actual Daily ManPower for Classification
Classification ManPower Needed per Task
Desired Daily ManPower for Classification
Classification ManPower Needed per Error
Development Productivity
Quality Assurance
Rework ManPower Needed Per Average Error
Daily Manpower Allocated For QA
Detected Errors in Testing
Maximum Daily ManPower for Classification
Error Scape
Daily Manpower Allocated For Rework
Error Detection in Testing
Testing
Rework
Bad Fixed Errors
TestingMapower NeededPerTask
Total Error Dencity in Testing
Undetected Errors
Daily ManPower Allocated for Testing
B1
B2
B3
B4
B5
B6
B7
R1
R2
R3
R4
R5
R6
R7

Figure 44 New loops are provided by Enhanced testing process

The most significant loops that I have chosen to describe from this CLD are loops B1, R1, R5, B5 and R7.

B1: The first balancing loop of this system concern percent of job actually worked, bad fixes, Errors density, detected errors in testing, rework considerable errors after classification and the detected errors waiting for rework.

When software is being developed, the percent of job actually worked increase. The increase signals that the project is moving from the design-stage to the coding stage, and the rework manpower needed per average error decreases. The relationship between these two factors is therefore negative. When the average manpower needed for rework decreases the rework rate increases. Since more tasks can be reworked per manpower allocated, the same amount of staff is able to process and rework more errors. However when the rework rate increases the chances for bad fixes increases.

Bad fixes occurs when the rework staff erroneously provide a solution to a problem that generates a new problem. This happens for different reasons such as motivational factors or stress. With more bad fixes, more errors are undetected within the tasks, and sent for testing. This leads to an increase in undetected errors. Subsequently an increase in the total error density in testing follows, and requires more testing manpower per task. More manpower needed per average error decreases the rate of testing performed by the testing staff. Less testing leads to less error detection in testing. Subsequently, fewer errors are detected in testing.

When fewer errors are detected, the desired daily manpower for error classification decreases. The decrease in desired manpower for error classification provides a decrease in the actual manpower for error classification. When the actual manpower for error classification decreases, and with less manpower, the rate of the potential errors classification decrease as well. This leads to a decrease in amount of rework considerable errors after classification stage, and the detected errors waiting for rework decreases as well. Schedule pressures will drop when the amount of detected errors waiting for rework are low and (this leads to a decrease in) daily manpower allocated for software development. The loop is closed when the daily manpower for software development decreases the software development rate and the percent of job actually worked decreases.

R1: The first reinforcing loop of this system concerns the software development, quality assurance, escaped errors, error density and error detection in testing, rework considerable errors after classification and detected errors waiting for rework.

When software is being developed and the project has started, the quality assurance effort starts as well. The more quality assurance activities in action, the less amounts of errors escapes detection. When fewer errors escape detection from the quality assurance effort, less undetected errors slip by to testing. The total error density for errors in testing increases when more errors are detected by the QA-effort. The total error density in testing influence the testing manpower needed per task. When the density of errors decreases, the manpower needed per task decreases as well since one task contains strings of codes that have errors. When the effort needed per task decreases the overall rate of testing increase as a result.

The increased rate of testing leads to higher error detections in testing. Higher detections leads to a higher amount of errors that are detected in testing, and the management's desired daily manpower for error classification is adjusted upwards. The adjustment leads to more actual manpower for error classification process, and the rate of potential error classification rate increases. The increase in potential error classification, increase the amount of rework considerable errors classified by the testing staff. This leads to more detected errors waiting for rework, and schedule pressure increases as a result. Increased schedule pressures, causes an adjustment from management, and daily manpower allocated for software development increases as well. The loop is closed when this increase in daily manpower allocation for development causes an increase in software development rate.

R5: The fifth reinforcing loop of this system concerns percent of job actually worked, the daily manpower allocated for rework, the daily manpower allocated for testing, the allocation of manpower to classification process, rework considerable errors after classification and the detected errors waiting for rework.

When the software project is underway and progressing, and the percent of job actually worked increases. This increase leads to less effort needed to rework an average error since project is moving toward the coding phase of development. This causes an adjustment from management, and daily manpower allocated for rework decreases as well. As the rework team belongs to the production effort, the allocation of manpower to rework limits the available staff for testing, as the testing staff is drawn from the production staff.

In this scenario, the daily manpower allocated for rework decreases. This leads to a subsequent increase in daily manpower allocation for testing. So, the relationship between the daily manpower allocated for rework, and the daily manpower allocated for testing is negative. The increase in daily manpower allocated for testing, leads to an increase in the maximum daily manpower allocated for the classification stage of the testing process. With more staff available for classification, the actual daily manpower for error classification logically increases. The potential rate of error classification increases as a result and the amount of the rework considerable errors after classification increases. It leads to an increase in the amount of the detected errors waiting for rework as well. Schedule pressures increases as more detected errors are waiting for rework and this causes an adjustment from management, and daily manpower allocated for production and development increases as well. The loop is closed as this increase leads to more software development and percent of job actually worked. The loop managed the allocation of manpower between the production efforts that are development, rework and testing based on the testing effort and the detected errors awaiting rework.

B5: The fifth balancing loop of the system concerns the percent of job actually worked and the rework-considerable errors after classification.

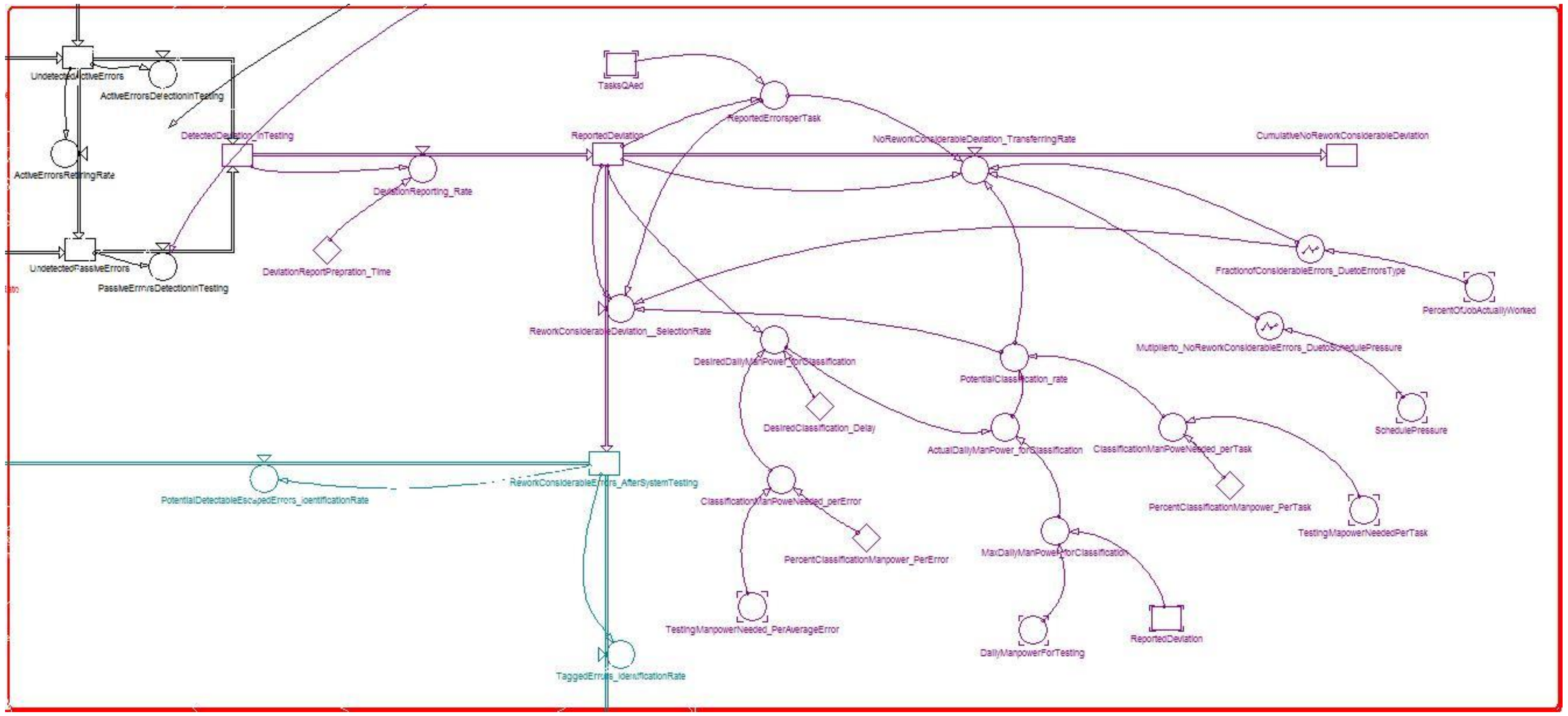
As the software project commences, the percent of job actually worked increases. The percent of job actually worked determines the stage of production for the software. The first 50% of the job is the design-phase of development, while the last 50% is the coding phase. When the percentage of job actually worked increases, the project moves from design-phase to coding phase. There are two key factors here that effect the selection of rework considerable errors during classification stage. Firstly, coding errors are by nature easier than design errors to detect, and fix in the previous rework stage of the production process. Secondly design errors are more significant errors as they cause and regenerate several coding errors along the progress of the project. While project is in design phase and errors are designs errors, the majority of them need serious consideration and rework. Therefore, when the project progresses and percent of job increases, the amount of rework considerable errors classified by testing decreases due to the fact of different error-types during the development progress. Since there at this point are more coding errors, the grave design errors that need rework are diminishing. With less rework considerable errors after classification, the detected errors waiting for rework decreases as well. This leads to less schedule pressures, and the daily manpower allocated for software development is decreases as well by managerial adjustment. This leads to less software developed, which halts the progress of the project and the percentage of job actually worked and the balancing loop is closed.

R7: The seventh reinforcing loop of the system links the percent of job actually worked to development productivity, the classification manpower needed per task and the potential rate for errors classification.

When the project progresses and the percentage of job actually increase, the software project is moving from design to coding phase of development. When the project has been running for a while the development staffs productivity increases. Due to learning, an increase in workforce experience and easier coding tasks as design tasks require more effort than coding tasks. As a result the staff is more effective and productive. With an increase in the development productivity, the classification manpower needed per task decreases. By this decrease in effort needed to classify a task, the potential rate of errors classification increases since the same amount of staff and effort can classify more tasks.

With increased in the potential rate of errors classification, the rework considerable errors after classification follows suit. With more amounts of rework considerable errors after classification, the detected errors waiting for rework increases as well. This leads to an increase in schedule pressure, and the upward managerial adjustment in daily manpower for software development. With more manpower allocated to production, the software development rate increases. The loop closes as the increase in software development leads to an increase in percent of job actually worked.

The stock and flow diagram for this new testing-structure is presented below.



UndetectedActiveErrors
ActiveErrorsDetectionInTesting
ActiveErrorsRetiringRate
UndetectedPassiveErrors
PassiveErrorsDetectionInTesting
DetectedDeviation_InTesting
DeviationReporting_Rate
DeviationReportPrepration_Time
TasksQAed
ReportedErrorsperTask
ReportedDeviation
NoReworkConsiderableDeviation_TransferringRate
CumulativeNoReworkConsiderableDeviation
ReworkConsiderableDeviation__SelectionRate
FractionofConsiderableErrors_DuetoErrorsType
PercentOfJobActuallyWorked
MultiplierTo_NoReworkConsiderableErrors_DuetoSchedulePressure
DesiredDailyManPower_forClassification
PotentialClassification_rate
DesiredClassification_Delay
SchedulePressure
ActualDailyManPower_forClassification
ClassificationManPoweNeeded_perTask
ReworkConsiderableErrors_AfterSystemTesting
PotentialDetectableEscapedErrors_identificationRate
ClassificationManPoweNeeded_perError
PercentClassificationManpower_PerTask
TestingMapowerNeededPerTask
PercentClassificationManpower_PerError
MaxDailyManPower_forClassification
TestingManpowerNeeded_PerAverageError
DailyManpowerForTesting
ReportedDeviation
TaggedErrors_IdentificationRate

Figure 45. Stock and flow for the enhanced testing structure

The first stage of this new structure is the stock of detected deviations in testing. Escaped errors from the quality assurance effort or bad fixes in the rework effort, allows some active and passive errors to stay undetected and transfer to the testing phase as undetected errors. In the testing process, pieces of software is gathered and tested by the testing staff. The first stage of this process is to test the software and detect deviations. Deviations are errors or faults in the software causing unwanted shortcomings or malfunctions. When such deviations occur, the testing staff reports the nature of the deviation on such tasks, and the tasks are sent to the next stage of classification. The transfer rate for tasks between testing and classification rests on the report preparation time. In my model this is assumed to be 3 days.

When a task has been sent to classification due to deviations or with no errors detected, the assigned testing staff begins the process of further testing the software. There are two classifications for a piece of software that has been tested. They can either be branded as "No rework required" or "Rework considerable". No rework required tasks hold no considerable deviations and they do perform the assigned function. Sometimes they do perform with minor shortcomings or flaws. If there is any end-user testing effort considered for the project, these tasks will be sent to the client review. Here the end-user can determine if flaws or the minor shortcomings are considerable enough to degrade the software's implementation or user-experience.

The second classification is "rework considerable". These tasks are pieces of software that do not function according to the design. These tasks contain design-errors and severe coding errors that cause major and considerable malfunctions and integration problems. Such errors are appearing in the test-effort when they undetectably slip by quality assurance or are badly fixed by the rework team. These tasks are gathered and sent to the re-capturing process through the identification stage in order to have them properly identified.

The factors and processes above are presented as the enhanced testing system in the model. The outflows for the stock of "Reported deviation" that are waiting classification are "No rework considerable deviations" and "Rework considerable deviations". Both these outflows hinges on the daily manpower allocated to testing, the testing manpower needed per average task, the manpower needed to test per average error, the reported errors per task, schedule pressure and the percent of job is actually done.

The first outflow that is “Rework considerable deviation selection rate” that is calculated as follows:

*Rework Considerable Deviation Selection Rate = (MIN ((Potential Classification rate * Reported Errors per Task), Reported Deviation / TIMESTEP)) * Fraction of Considerable Errors Due to Errors Type*

As is shown in the formula above the outflow depends on the amount of reported deviation waiting for classification. Secondly, the selection rate is determined by the reported errors per task. This variable provides an average density of reported errors per task that is calculated as a function between the “Reported deviation” in testing phase and the “Tasks quality assured” after development process. It is a maximum function and it is calculated as follows:

Reported Errors per Task = MAX (Reported Errors Waiting Classification / (Tasks QAed + 0,0001); 0).

The third element stems from the “Potential classification rate” that exists in the testing process. There are two determinants for the potential classification rate. The first is the “Classification manpower needed per task” and second one is “Actual daily manpower for classification”. This “Potential classification rate” is then linked to the “Rework considerable deviation selection rate” and functions as the third determinant for this outflow. The variable it is calculated as follows:

Potential Classification rate = Actual Daily Manpower for Classification / Classification Manpower Needed per Task

To determine the “Classification manpower needed per task” there are two significant variables. The first is the “Testing manpower needed per task”. The formulation of this variable was given in the earlier enhancement regarding the manpower effort needed per task for the testing process. In short, the manpower needed per task hinges on error density, development productivity and the testing manpower needed per error. The effort needed to classify a task is much less than the effort needed to test a task. Therefore, “Classification manpower needed per task” is determined by the same logic as “Testing manpower needed per task” under the effect of “Percent classification manpower per error”. In my model this percentage is assumed to be 10%. This variable is calculated as follows:

*Classification Manpower Needed per Task = Testing Manpower Needed Per Task * Percent Classification Manpower per Task*

The "Actual Daily Manpower for Classification" is a minimum function between the desired and the maximum daily manpower for classification. These variables provide the first half of the calculation for the potential classification rate, namely the amount of manpower available for classification. The variable takes the minimum value between the desired and maximum value of daily manpower to control for over-allocation of manpower to classification stage compared to the desired manpower.

Actual Daily Manpower for Classification = MIN (Desired Daily Manpower for Classification, Max Daily Manpower for Classification)

The "Desired daily manpower for classification" is determined by the "Classification manpower needed per error", "Reported Deviation" and the "Desired classification Delay". In my model this delay is assumed to be 3days. Desired daily manpower for classification provides the first part of the calculation for "Actual daily manpower for classification".

*Desired Daily Manpower for Classification = (Reported Deviation * Classification Manpower Needed Per Error) / Desired Classification Delay*

The effort needed to classify an average error is much less than the effort needed to test an average error. Hence, "Classification Manpower Needed per Error" is determined by the same logic as "Testing Manpower Needed per Average Error" under the effect of the percent effort for classification called "Percent Classification Manpower per Error". This variable is calculated as follows:

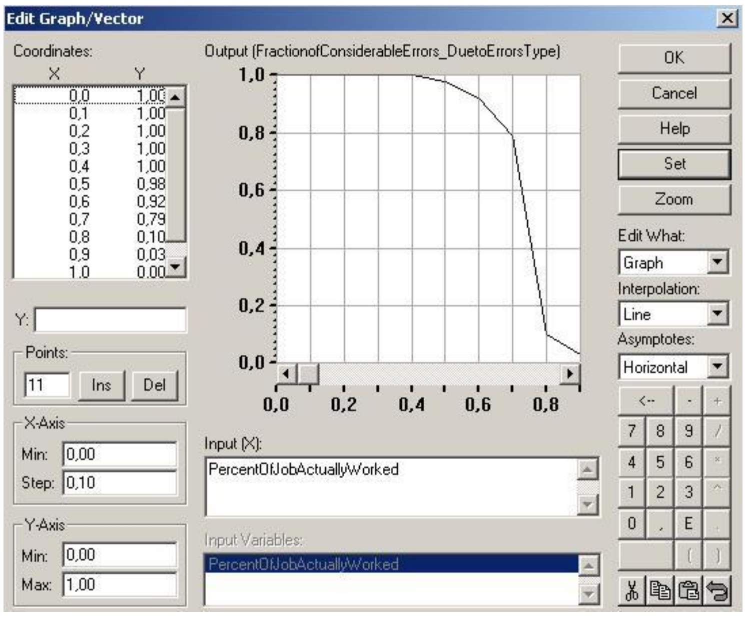
*Classification Manpower Needed per Error = Testing Manpower Needed Per Average Error * Percent Classification Manpower per Error)*

The "Maximum daily manpower for classification" rests on the "Daily manpower allocated for testing" and controlled by the stock of "Reported Deviation". Daily manpower for testing is determined by management's allocation of the total effort to systems testing. An if-function is applied here to control the allocation of manpower to the classification stage as soon as reported deviations are prepared for classification. This variable is calculated as follows:

Maximum daily manpower for classification = IF (Reported Deviation >0; (Daily Manpower for Testing); 0)

The fourth and final determinant for the out flow of “Considerable deviation selection rate” is the percent of job actually worked through the “Fraction of considerable errors due to errors type”. From previous explanations, I explained how the percentage of job actually done determines the stage of production for the software. The progress represents the movement of the production phase from design to coding. The first 50% of the job is the design-phase of development, while the last 50% is the coding phase. There are two key factors here that effect the selection of rework considerable errors during the classification stage. Firstly, coding errors are by nature easier than design errors to detect, and fix in the previous rework stage of the production process.

Secondly design errors are more significant errors as they cause and regenerate multiple coding errors along the progress of the project. While the project is in design phase and errors are designs errors, the majority of them need attention and concentration. Therefore, when the project progresses and the percent of job increases, the amount of rework considerable errors classified by testing decreases due to the effect of different errors type along the development phase. Since at this stage there are more coding errors, the grave design errors that need rework are diminishing. Hence, this variable is the final determinant to the flow between “Reported Deviations” waiting for classification and “Rework Considerable Deviations” sent to the identification stage. This multiplier follows nearly the same logic and pattern as the percentage of active errors compared to amount of job is worked in the base model (Abdel-Hamid, 1991: p.112). During the design phase the multiplier holds a factor of one, as the deviations sent for classification are a result of design errors that are significant errors and needs rework consideration. When the project reaches the coding-phase, the amounts of errors are increasingly coding errors and the need for rework consideration becomes less as the project progress towards the end. This multiplier is depicted below:



Output (FractionofConsiderableErrors_DuetoErrorsType)

Coordinates:

X	Y
0,0	1,00
0,1	1,00
0,2	1,00
0,3	1,00
0,4	1,00
0,5	0,98
0,6	0,92
0,7	0,79
0,8	0,10
0,9	0,03
1,0	0,00

Points: 11

X-Axis Min: 0,00 Step: 0,10

Y-Axis Min: 0,00 Max: 1,00

Input (X): PercentOfJobActuallyWorked

Input Variables: PercentOfJobActuallyWorked

Edit What: Graph

Interpolation: Line

Asymptotes: Horizontal

Figure 46. Fraction of considerable errors due to errors type

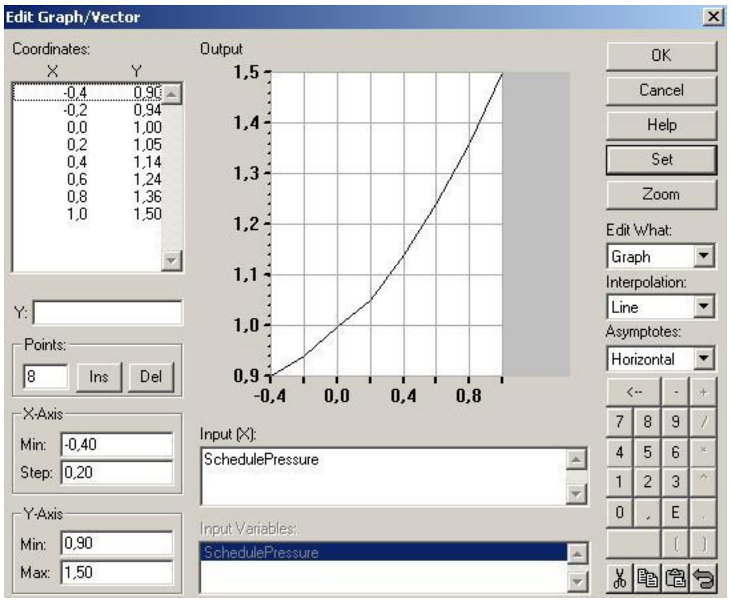
The second outflow for the stock of “Reported deviation” waiting classification is “No rework considerable deviations”. This outflow also hinges on the daily manpower allocated to testing, the manpower needed to rework per average task, the manpower needed to rework per average error, the reported errors per task, schedule pressure and the percent of job actually done. This outflow is calculated as follows:

*No Rework Considerable Deviation Transferring Rate = (MIN ((Potential Classification rate * Reported Errors per Task); Reported Errors Waiting Classification / TIMESTEP)) * (1-Fraction of Considerable Errors Due to Errors Type) * Multiplier to No Rework Considerable Errors Due to Schedule Pressure.*

As is shown in the formula above, the outflow depends on the amount of reported deviation waiting for classification. Secondly, the selection rate is determined by the reported errors per task. The third element stems from the “Potential classification rate” that exists in the testing process. The fourth determinant for the out flow is the percent of job actually worked through the “Fraction of considerable errors due to errors type”. During the presentation of the first outflow, these four determinations where explained in details.

The fifth determinant is a multiplier due to schedule pressures. Schedule pressures have been identified in the literature as a base for productivity challenges, error generation and bad fixes.

High schedule pressures lead to poor performance by the staff or aggressive decision making by the managers. The multiplier is depicted below:


Edit Graph/Vector

Coordinates:

X	Y
-0,4	0,90
-0,2	0,94
0,0	1,00
0,2	1,05
0,4	1,14
0,6	1,24
0,8	1,36
1,0	1,50

Points: 8

X-Axis Min: -0,40 Step: 0,20

Y-Axis Min: 0,90 Max: 1,50

Input (X): SchedulePressure

Input Variables: SchedulePressure

Edit What: Graph

Interpolation: Line

Asymptotes: Horizontal

Figure 47. Multiplier to no rework considerable deviations due to schedule pressure

From the diagram, we see that an increase in schedule pressures leads to a relative increase in the multiplier to no rework considerable deviations. As the project moves towards the scheduled completion date, the amount of pressure rises accordingly. Pressure situations leads to more aggressive decision making by management such as reducing the effort related to the quality of the product. Under stress situation some considerable deviations can be skipped from further consideration and rework. Instead they are chosen as no rework considerable deviations.

This is consistent with the description of the schedule pressures in the literature. From the Abdel-Hamid thesis, empirical examples depicted how quality assurance could be halted when a project was severely behind schedule (Abdel-Hamid, 1991: p.83). The same logic follows the choice to reconsider the standard for a "severe" error. When a software project is running behind schedule, management will make risk-calculations where the pros and cons of current practice are discussed. If the costs of not delivering outweigh the costs of lowering the standard for "severe" errors, this is an option to ensure delivery on time. It will however, affect overall quality.

From the base-model of Abdel-Hamid we learned that under schedule pressure the allocation of manpower to the quality assurance effort was reduced by 0.5 (Abdel-Hamid, 1991: p.73). This translates to a potential reduction of 0.5 for the overall product quality, as an important effort ensuring software quality is reduced significantly. If we translate this willingness to adjust on the quality side for schedule pressures, I can adopt a similar relationship for deciding rework-considerable errors. In the normal situation the multiplier holds no effect. In this case, any deviation that infringes on the preset standards is considered rework-considerable errors. However, as schedule pressure increases, the priority is shifted to finishing on schedule. In such a scenario, the willingness to debate or reconsider standards increase. This leads to a situation where a piece of software that would under no pressure be sent to rework, is considered as no-rework considerable instead.

I create here an example to illustrate my point: The preset standard for the project held 6 criteria that an application should pass in order to be a no-rework considerable deviation. During a time of schedule pressure they find pieces of software that fulfill 5 out of 6 criteria. In order to finish on time, the standard is lowered to 5 criteria rather than 6. This enables more software to be considered as no-rework considerable and the project picks up speed. Management can reconsider standards in order to make the project run faster. When pressures are high, the willingness to reconsider standards to get back on track increases in this fashion. It is the same logic as suspending quality assurance or cancelling it to finish on time. Suspending quality assurance is also a form of reconsidering standards. This reaches a top of 0.5 for the multiplier, mirroring the effect of schedule pressure on manpower allocated for quality assurance. Therefore, I have chosen to assume a multiplier ranging from 1 with no pressure to 1.5 with full schedule pressure. The tasks with no rework considerable deviations represent the assigned function, but with some minor shortcomings or flaws. If there will be an end user review considered for the produced software, these tasks will be transferred to client review stage later on where the end-user can determine if flaws or the minor shortcomings are considerable enough to degrade the software’s implementation or user-experience.

Chapter 7: Model Validation and analysis

7.0 Introduction

Model validation is an important endeavor for all branches of model-based analysis, and especially in the System Dynamics methodology (Barlas, 1996). All models are simplified representations of real-world mechanisms and systems that govern the observed behavior we wish to investigate. It is therefore of vital importance that the model we construct can be validated through empirical, logical and numerological tests to build confidence in results and system.

The concept of validation in system dynamics is related to the general scientific question of theory-validation. System dynamic models claim to be causal models, and therefore any evidence that refute the causality of equations inside the model refutes the model as a whole. The output from the model may very well provide an accurate description of the observed behavior, however if the causality is erroneous the results are without validity. Every system dynamic model contains a theory on how the observed structure functions. This theory must then be tested and validated to build confidence in the results. Hence, the concept of model-validation and the general philosophy of scientific issues are highly related to one-another (Barlas, 1996).

Sterman argues that no model can ever be fully validated, as every model is a limited and simplified representation of reality (Sterman, 2003: p.848). The issue of model validation is therefore complex and challenging. Forrester and Senge argue that validation is a process were one build confidence in the model’s soundness and applicability. There are several techniques and tests that allow us to build increased confidence in our model in terms of both logic and numerology (Forrester & Senge, 1980). There are two main areas of model-validation that provide the battery of total validation-tests applicable to a system dynamics model. The first area is the structure aspect of the model, while the second focus on the structural behavior variables inside the model (Barlas, 1994).

I will now proceed with the validation-effort for my model. The techniques applied here follow the recommended tests for both the structural aspects, and for the internal dynamics of my model. The tests are provided by Sterman and Forrester & Senge (Sterman, 2003: pp. 858-891), (Forrester & Senge, 1980).

7.1 Direct structure test

The direct structure test assesses the validity of the model structure. This is achieved through comparing the model structure directly with knowledge about the real system. This involves taking each relationship individually and comparing it with available knowledge about the real system. Barlas argues that the structure test can be divided into an empirical approach and a theoretical approach (Barlas, 1996: pp.189-190). For the empirical approach one investigates the empirical description of the problem that is modelled and the system it represents. The model and the relationships within must have real-world counterparts (Sterman, 2003).

The theoretical approach of the direct structure test looks at the parameters and the equations inside the model. These parameters must be equal to real-world relationships, and follow real-world logic. It is important to assess that the numerical material is sufficient, and that the values provided are reflecting the true values reported in the empirical material (Barlas, 1994: p.190).

Finally, Forrester and Senge suggest that a unit consistency test should be performed under the direct structure test as well (Forrester and Senge, 1980). This test focuses on the equations in the model, and the units that are derived from the calculations. The equations should not provide erroneous units, ill-logical consequences (such as water running up-hill), and follow the real-world rules of nature (Sterman, 2003).

Common for all these types of tests is the fact that there is no simulation done for them. This is an empirical and theoretical discussion of the model structure, rather than a simulation-test. Hence the form and procedure followed varies from case to case, and there is no strict formal blueprint to follow. In my paper I will provide both an empirical, and a theoretical presentation of my model. Then I will perform a unit consistency test.

7.1.1 Structure test

The model that I am presenting in my thesis is based on Abdel-Hamid's original pioneering effort into the software crisis. From my literature review, I have highlighted the position of this thesis in the scholarly tradition of the system dynamics field. The model has been used as a platform for additional studies into the problem of software production, cost-overruns, delays and poor software quality. Hence, the base of my work rests on a solid empirical foundation provided by the strong model of Abdel-Hamid.

The concept of the software crisis and its continued presence in modern times has been clearly documented as well. Through the article of Robert Charette, the empirical descriptions of Jones and Bonsignour plus the reports from both ESSU and the DNV I have shown the existence of the software crisis in the late 2000nds and that it follows the same remarkable pattern as described by Abdel-Hamid in 1991. This strengthens the validity of the concept further.

In my study, there are five enhancements and one policy structure. The first enhancement is a testing-regime that is continuous and consists of multiple stages. The software will now be tested as applications are finished, and milestones achieved. This also bridges the gap between the testing-effort and the rework effort found in the original model. Another enhancement is the capture-recapture effort that ties quality assurance and testing together. This effort is a new approach into defect-removal that combines the strengths of both testing and quality assurance together. Through the policy structure I incorporate a client-review effort that allows the client to be involved in the process. This allows the client-side that was previously excluded from the model to participate. It is important that these enhancements and the policy structure also have real-world counterparts, and that they are indeed logical for a real-world scenario.

The concept of testing is well documented in Jones' and Bonsignour's work. The authors have several years of experience in the American and the European software development environment. Their empirical data stems from cooperation with large software companies such as AT&T and IBM. From the book "The Economics of Software Quality" it becomes apparent that the testing effort is used as a tool for defect removal in all software projects. The type of testing varies, and the amount of testing varies. The determining factor for testing is the project's size. In the original model, the testing-stage consisted of one effort. This was the final systems test at the end of a project. From Jones' and Bonsignour's empirical explanation of testing, it is clear that no medium-sized modern software project can utilize only one testing stage. Today's software is far more complex than the software at the end of the 80s and early 90s. They describe a development where testing is used in several stages to provide milestones, and to test applications as the software is developed. This is done in order to rework errors before the final systems test. Errors found in the last systems test are much more expensive to fix by as much as ten times (Jones and Bonsignour, 2012: pp.280-285). The empirical description provided here strengthens the validity of my testing-regime. It is important that the model is able to reproduce empirical descriptions of the effort in real-world. If I had kept the original one-test regime, it would have breached the real-world reality of a modern medium-ranged software project.

In my enhanced model, I therefore propose a continuous system of testing stages. In the data on American software project, it is apparent that several testing stages are the norm. Secondly, for a medium software project it is a necessity. The numerical evidence is provided in the table below.

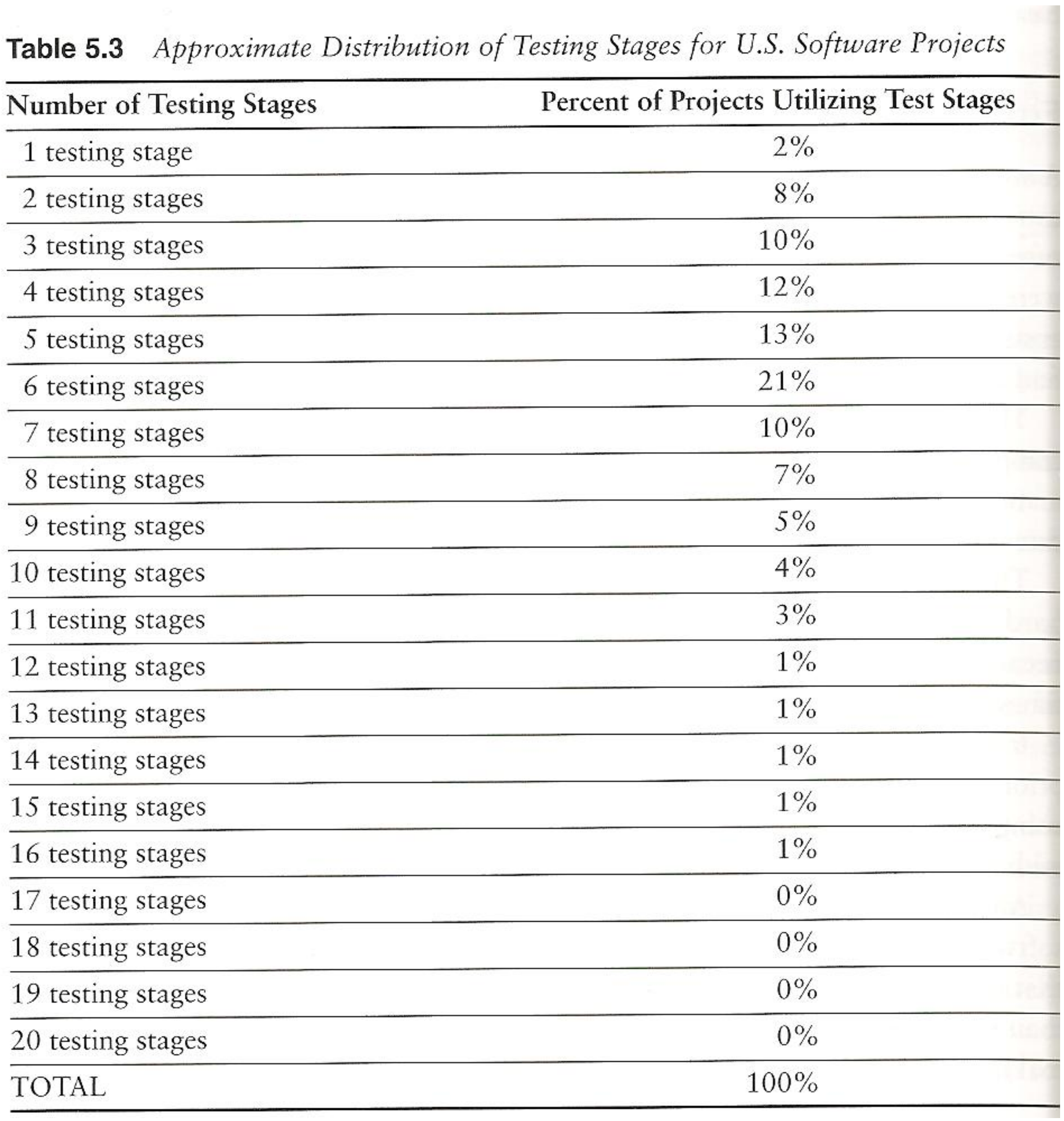


Table 5.3 *Approximate Distribution of Testing Stages for U.S. Software Projects*

Number of Testing Stages	Percent of Projects Utilizing Test Stages
1 testing stage	2%
2 testing stages	8%
3 testing stages	10%
4 testing stages	12%
5 testing stages	13%
6 testing stages	21%
7 testing stages	10%
8 testing stages	7%
9 testing stages	5%
10 testing stages	4%
11 testing stages	3%
12 testing stages	1%
13 testing stages	1%
14 testing stages	1%
15 testing stages	1%
16 testing stages	1%
17 testing stages	0%
18 testing stages	0%
19 testing stages	0%
20 testing stages	0%
TOTAL	100%

Figure 48. Approximate distribution of testing stages for U.S. software project

From this table we clearly see that having only one testing-stage would be problematic. Only 2% of all software projects use such a regime, and the model would fail to cover the reality of modern software projects. Hence, I propose a continuous system which is the most common testing effort. 21% of all software projects in the US utilize six stages (Jones and Bonsignour, 2012: p.341).

The "capture re-capture" effort is described empirically by Jones and Bonsignour (2012), Scott and Wohlin (2008) and Petersson and Wohlin (1999). The capture re-capture idea is borrowed from the realm of biology. In order to determine the true size of a population of a species, a sample of animals are trapped and tagged. Later they are set free and allowed to roam for a given time period. Then the scientists will recapture the species in the same area, and count the number of tagged animals they catch. This allows the research-team to statistically calculate the true size of the species based on the number of tags.

In the software project environment this idea is transferred onto errors. When an error is found by the quality assurance team it is sent to rework. Then after the rework-teams have fixed the error, it is sent back to quality assurance and tagged. The task containing the fixed error is then passed on into the system, until it ends up in the testing effort. This procedure allows management to calculate the true size of the error-population. It also reveals where an error stems from. A tagged error is a bad-fix, while an untagged error is an error that escaped detection in the quality assurance effort. The capture-recapture effort has been applied in software projects at for example IBM (Jones and Bonsignour, 2012). The fact that three sources describe my capture-recapture effort in such a manner, and that it has been deployed in the real world strengthens the validity of my model. It is both described and deployed in a modern software development environment.

My policy structure is the client-review stage. According to Hjertø, it is very important to include the client for the software project (Hjertø, 2003). In the original base-model, the client was only included in the pre-production planning stage. In this stage the clients specify his needs, and the software blueprint is planned. When the software production commences, the client is not included at any other stage in the base-model. In my literature review, the traditional TQM thinking stressed the importance of the client. To achieve the needed quality, the end-user of any product should be heard. It is the end-user that determines a product’s functionality, not the designer that made it. This is a very important concept to understand. The product could be made according to all engineering standards and vices, but if it’s over-engineered and impractical for the end-user it is still a failed product (Deming, 1982). This will lead to unhappy customers and angry clients.

In the field of software production the client is incorporated as a resource for testing and pre-completion validation. It is increasingly normal to hold end-user testing or beta-testing of a program for potential buyers or the clients. This gives management an additional channel for information on the progress of the software, and if it is following the standards set in the pre-planning stage. According to Jones and Bonsignour, the client is a participant during the development of the software programme. The program is tailored to the client's needs, and it's the client who knows the real-world environment the software will be operating in. These empirical descriptions of the client as a participant increase the validity of my client-review. The client-review is a series of tests that allow the client to familiarize with the applications of the programme, and if these covering his needs. This system is similar to the description provided by Jones and Bonsignour of end-user testing (Jones and Bonsignour, 2012).

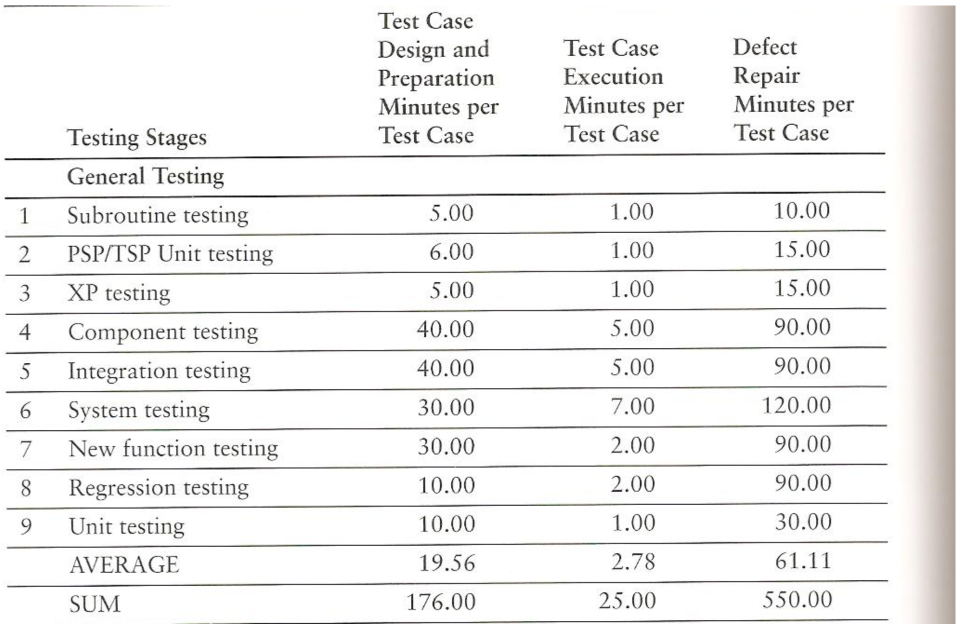
From these empirical descriptions I argue that my model and enhancements have a solid empirical foundation. They have real-world counterparts, and the relationships described match the relationship inside my enhanced model.

7.1.2 Parameter test

The parameter test is the second test I will deploy that belongs to the structure analysis test. Unlike the previous test, the focus will now be on the parameters set in the model. It is important that the variables and calculations follow real-world data. One should to the highest degree possible apply parameters that have a foundation in empirical material. In this test I will present a few parameters in order to establish this concept. I will also provide direct numerical material from the literature I deploy for my model. Due to the size and scope of this model, I will not be able to present every parameter here. I will however, present material that is relevant for the changes that my enhancements have made.

The first parameter is the percent bad fixes. In the original model, this numerical value was a constant at 7.5%. This constant derived from empirical data of the time, and the variation reported was 6.5 to 10%. In the work of Jones and Bonsignour however, we found a slightly different picture. The data compiled by the authors extend over 20 years of research, and the data gave new information. The industry average is still 7% as in Abdel-Hamid's case. However, there was a bigger difference between the successful and the failed projects. The best organizations had an average of only 2% bad fixes after production, while the worst cases provided 20-25% of bad fixes (Jones & Bonsignour, 2012). With these data, I determined to make the constant into a structure. My variable is determined by several factors during the software project, and the percentage of bad fixes holds a variation between 3% and a high-score of in the 20%s. From the empirical data-material this is within the range of bad fixes in a real-world software project. Hence, the validity of this change to the model has been strengthened. In addition I have been able to provide a more realistic process of production and bad fixes.

The second parameter here concerned the effort to testing compared with rework. In the base-model the effort to rework an error was set in a multiplier that ranged between 0.6 and 0.3. For the testing effort I used the logic behind the difference in effort due to error type, but reduced the numerical value to 0.3 and 0.2. The data provided by Jones and Bonsignour revealed the effort in minutes per error for testing and for reworking. It is clear from the table below that the minutes for testing a case, are far less than for reworking an error. Therefore I chose a smaller parameter for this multiplier.

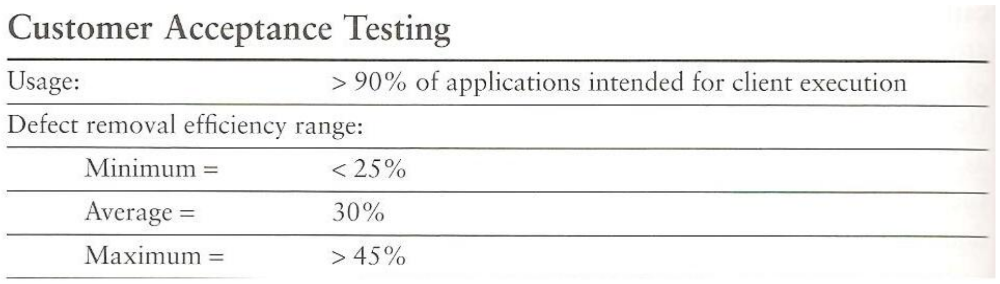


	Testing Stages	Test Case Design and Preparation Minutes per Test Case	Test Case Execution Minutes per Test Case	Defect Repair Minutes per Test Case
	General Testing			
1	Subroutine testing	5.00	1.00	10.00
2	PSP/TSP Unit testing	6.00	1.00	15.00
3	XP testing	5.00	1.00	15.00
4	Component testing	40.00	5.00	90.00
5	Integration testing	40.00	5.00	90.00
6	System testing	30.00	7.00	120.00
7	New function testing	30.00	2.00	90.00
8	Regression testing	10.00	2.00	90.00
9	Unit testing	10.00	1.00	30.00
	AVERAGE	19.56	2.78	61.11
	SUM	176.00	25.00	550.00

Figure 49. The effort in minutes per error for testing stages
Source: Jones and Bonsignour, 2012: p. 338

The parameters and numerical data available strengthen the validity of my testing regime. The number of stages and the principle is common. Secondly the effort needed for the testing-stage is based on real-world numerical values.

The client review that is added as a policy structure is based on the notion that the client is an important source for software quality. In my policy structure I have built a model structure where the client and development-staff have a joint-effort. In this effort clients test applications that will be part of the finished software, while the designated staff from provides a presentation and clarify the feedback from the client. This idea is also based on empirical numerical data. The usage of acceptance testing is fairly broad. The data from jones and Bonsignour suggests that over 90% of applications intended for client execution are subjected to an acceptance test. This is provided in the figure below.

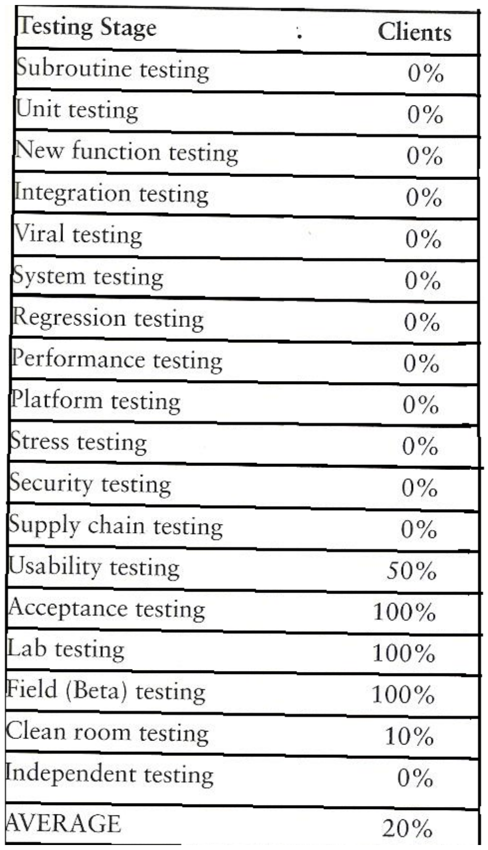


Customer Acceptance Testing

Usage:	> 90% of applications intended for client execution
Defect removal efficiency range:	
Minimum =	< 25%
Average =	30%
Maximum =	> 45%

Figure 50. Customer acceptance testing,
Source: Jones and Bonsignour, 2012: p. 335

The client is kept as an outside observer and tester in this case. They are not invited into other testing-efforts during the software project. This is also provided by numerical data of Jones and Bonsignour. The table below shows the client-participation in percentage during different testing-efforts. It is apparent that the client is only involved in beta-testing, acceptance testing, lab testing and usability testing. In my model an acceptance test is deployed. The client manpower, client manpower productivity and client effort is needed to review the software, but the control over such effort lies in the hands of the client himself. Hence, these factors will be exogenous in the model. This is explained in detail in the next chapter.



Testing Stage	Clients
Subroutine testing	0%
Unit testing	0%
New function testing	0%
Integration testing	0%
Viral testing	0%
System testing	0%
Regression testing	0%
Performance testing	0%
Platform testing	0%
Stress testing	0%
Security testing	0%
Supply chain testing	0%
Usability testing	50%
Acceptance testing	100%
Lab testing	100%
Field (Beta) testing	100%
Clean room testing	10%
Independent testing	0%
AVERAGE	20%

Figure 51. Testing stages,
Source: Jones & Bonsignour, 2012: p. 338

The numerical data provided here in my parameter test shows that the inclusion of enhancements and their numerical value is supported by the real-world figures. It strengthens the reliability of my variables and the results they yield. Secondly the foundation for my model has been strengthened by both a theoretical discussion and numerical discussion of parameters. This concludes the parameter test.

7.1.3 Dimensional consistency test

Dimensional consistency is one of the most basic tests for model validation (Sterman, 2003: p.866). The unit of measures for each variable must be consistent with real world meaning. Hence, every equation must be dimensionally consistent without the inclusion of arbitrary scaling factors that have no real world meaning. Unit errors reveal important flaws in the understanding of the structure or decision process modeled (Sterman, 2003: p.866).

In order to pass this test, it is necessary to provide some variables from the sub-sectors I built for my enhancements, and show how they are unit consistent. The table below shows the unit consistency of some of my variables:

Unit	**Formulation**
Error	= Detected Errors Waiting For Rework
Task	= Cumulative Tasks Quality assured
Error/Task	=MAX (Detected Errors Waiting for Rework / (CumulativeTasksQAed+0,0001), 0)
Day	= NoConsiderableErrors_ReportPreperation_Time
Error	= NoReworkConsiderableErrors
Error/Day	= Non Rework Considerable Errors / No Considerable Errors Report Prepration Time
ManDay	= Cumulative Testing Man-Day
Tasks	= Cumulative Tasks Tested
Task/ManDay	= CumulativeTasksTested/(CumulativeTestingMD+0,001)

BadFixes Generation Rate	= (Detected Errors Rework Rate * Fraction of Bad Fixes) * Multiplier to BadFix Generation Due to Reworked Errors Density
Error per Task	= (Error per Task * dimensionless) * dimensionless
	= Error per Task

Rework Manpower Needed Per Average Error	= Nominal Rework Manpower Needed Per Error * Multiplier to Rework Effort Due To Detected Error Density * (1 / Multiplier Due To Manpower Productivity)
Man-Day per Error	= Man-Day per Error * dimensionless * (1 / dimensionless)
	= Man-Day per Error

Potential Tag Releasing Rate	= Actual Daily ManPower for Tagging / Tagging ManPower Needed Per Error
Error per Day	= Man-Day per Day / Man-Day per Error
	= Error per Day

Testing Mapower Needed Per Task = ((Testing Effort Overhead * DSI Per Task / 1000) *(1/Multiplier Due To Manpower Productivity)) + (Testing Manpower Needed Per Average Error * Total Error Density)

Man-Day per Task = ((Man-Days per KDSI * DSI Per Task) * (1 / dimensionless) + (Man-Day per Error * Error per Task)
= Man-Day per Task

Rework Considerable Errors Selection Rate = (MIN ((Potential Classification rate * Reported Errors per Task), Reported Errors Waiting Classification / TIMESTEP)) * Fraction of Considerable Errors Due to Errors Type

Error per Day = (MIN ((Task per Day * Erro0r per Task), Error / Day)) * dimensionless
= Error per Day

From the examples above I have shown some examples that describe the unit consistency of my variables, and that they hold real world dimensional consistency without parameters with weird names or strange combination of units.

Based on the explanations provided above, I argue that my model satisfy the requirements for the direct structure test. My variables have real world counterparts, and my aggregations are within realistic values based on empirical data from the literature.

7.2 Structure oriented behavior test

According to Barlas (1996), the second main category for the structural tests for model validation is the structured oriented behavior test. These tests are indirectly assessing the validity of the structure by applying certain behavior tests. These tests are all tests on behavior generated by the model, and the tests involve simulations. This is a key difference between the two forms of structure test, as the direct structure test is performed without any simulations.

The simulations that are run for structure oriented behavior tests are applicable to the entire model as well as isolated substructures. Barlas argues that these tests are strong behavior tests that can help uncover potential structural flaws (Barlas, 1996: p.191).

7.2.1 Boundary adequacy Test

The model boundary test asses the adequacy of the model boundary. In a system dynamic model all processes within the model should be explained endogenously. However, it is impossible to model every single real-world variable that will have an impact on the behavior of the system. It is therefore important to limit the model's scope and width in order to provide a clear model boundary. When such a boundary has been established, the adequacy of the boundary must be tested. If there are variables that influence the system exogenously outside the model boundary, these factors should be examined. If these variables are indeed influencing the model significantly and there is no logic for their exclusion of the model the boundary should be expanded to include them (Sterman, 2003: p. 861).

In my model the boundary of the base-model has been expanded in order to include the policy structure that is the client-review in the testing phase. The client-review is conducted in tandem with software production, and involves two teams of actors. The first team is allocated from the testing staff, provided by the software project management. The second team is allocated from the client, and consists of members from the client's agency, firm or institution. The model boundary runs vertically between the team allocated from testing and the team allocated by the client. There are two reasons for this boundary, one empirical and one numerical.

The empirical argument rests on the limitation of managerial responsibility in a software project. The aim of this model is to increase the understanding of challenges for managers in software development, and to find policies to improve software quality. Therefore all managerial aspects of manpower allocation and productivity are endogenously modeled. The client's position towards client-reviewing is outside the managerial area of responsibility. There is no empirical data that suggests a system where software production management allocates and control productivity for client personnel allocated to client-reviewing. The idea of end-user testing by the client is an opportunity for the client. How the client chooses to respond to such an opportunity is his own privilege. Hence, the model boundary includes every member of staff that is allocated by software management, but treats exogenously the staff allocated from the client's side.

The second argument is numerological. When we create the model boundary the productivity and allocation for the client is modeled exogenously. In order to control that these variables do not impact the behavior of the model directly, I performed a sensitivity test. If the model's behavior change pattern, or is significantly altered by changing the constant's value they are not appropriate to keep exogenously and should be included in the model. When I ran my test, the behavior-pattern of software development did not change. If the client dedicates 10 staffs or 2 staffs to client-reviewing, the variables change only in value and not in behavior. These two arguments strengthen the confidence in my model boundary. The important variables that influence production are endogenous, while aspects outside managerial control are exogenous.

<u>7.2.2 Integration method and choice of DT</u>

System dynamic models are continuous in time, and solved by numerical integration. The modeler must select numerical integration method and time steps that yield an approximation of the underlying dynamics accurate enough for the purpose (Sterman, 2003: p.872). The results from a model should never be sensitive to the choice of time-step. If the behavior of the model or the results changes significantly when the time-step has been halved, the model fails the integration error test. Wrong time-step or integration method may cause spurious dynamics to the model (Sterman, 2003: p.872).

In my model the original time-step was set to 0.5 days. The rule of thumb is to select a time step that is half of the shortest time-value of my model. The shortest value of the model is 1 day. The time-step was therefore set to 0.5 day. For the integration error test I set the time step to 0.25 days and 0.15 days. Neither reduction in time-step yielded any change in the model's behavior or results. The graphs below provide this assessment.

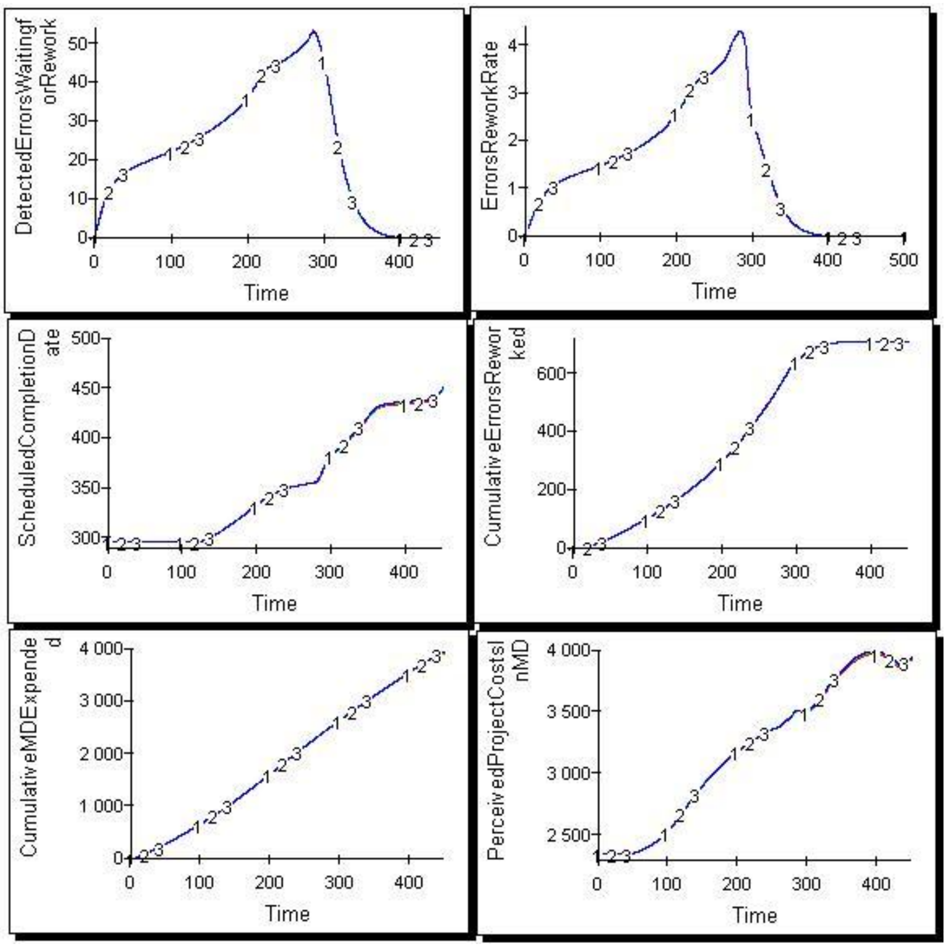


Figure 52. Some of the main variables' behaviors,
by running the simulation with different time steps

Run 1 is time step of 0.50 day, run 2 is time step of 0.25 day and run 3 is time step of 0.15 day

The graphs above all depict important stocks and variables in this model. I have run three simulations where the results from each run was saved and compiled into one graph. The first run had a time-step of 0.50 day. The second run had a 0.25 day time step, while the third simulation-run had a 0.15 day time-step. For the model to pass this test there cannot be any deviation between the values from each test. In the graphs above, we see that there are no deviations between each model run. As a result it is not possible to distinguish the different colored lines between each run. Hence the visible line have the markings of each model run on it, seen as numbers 1, 2 and 3 in the graphs. From this test I can conclude that my model passes the integration error test. This strengthens the validity of the behavior in this model, and that they are not a result of spurious effects.

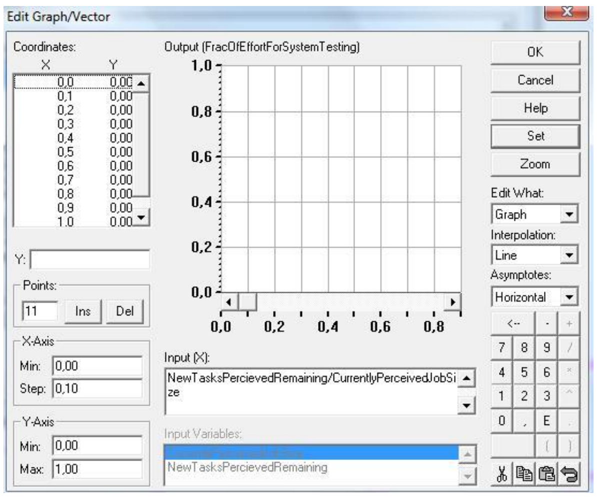
7.2.3 Extreme Conditions Test

In the extreme conditions test the equations in the model are subjected to control. The model should be robust in all conditions, and perform realistically under any circumstance. If there are errors or misconceptions in the equations extreme inputs or policies will create unrealistic effects (Sterman, 2003: p.169). One example of an unrealistic effect would be a bath-tub system with an erroneous delay. Imagine a policy of adding 100 billion liters of water to the bathtub, and the stock went to 100 billion liters by 0.1 seconds without any overflow. This would be a case of erroneous equations breaking the laws of physics. Stocks can never go negative as well. Inventories for example can only go to zero no matter how large the demand. (Sterman, 2003 : p.169).

In my model the software production hinges on allocation of manpower. It is therefore natural to initiate an extreme conditions test on this related variable. Software production can only operate when manpower is allocated to all the different efforts of the production cycle. Industries cannot operate if central resources such as labor are not allocated to key tasks.

In my extreme conditions test, I allocate no manpower to testing process. This should cause a situation where the entire testing effort and process stop, and all related stocks are constantly zero. If there still is testing process going on and is being conducted when no manpower has been allocated for it, my model fail the extreme conditions test. The results from the test are presented below:

The first step in this test is to set the effort for testing to zero, so that no manpower is allocated for this effort. In this model the manpower-allocation for testing hinges on a graph-function of the "Fraction of testing effort" variable. The first figure below shows the initialization of no manpower for testing. The fraction of testing effort is now held at zero and all the manpower is allocated to development. Even though tasks are being produced and accomplished, management does not perceive the need for testing-manpower. The result is a situation where no manpower is allocated and the testing process should not commence.



Output (FracOfEffortForSystemTesting)

Coordinates:

X	Y
0,0	0,00
0,1	0,00
0,2	0,00
0,3	0,00
0,4	0,00
0,5	0,00
0,6	0,00
0,7	0,00
0,8	0,00
0,9	0,00
1,0	0,00

Y:

Points: 11

X-Axis: Min: 0,00 Step: 0,10

Y-Axis: Min: 0,00 Max: 1,00

Input (X): NewTasksPercievedRemaining/CurrentlyPerceivedJobSize

Input Variables: NewTasksPercievedRemaining

Edit What: Graph

Interpolation: Line

Asymptotes: Horizontal

Figure 53. Fraction of effort for system testing

When the fraction of testing effort for systems testing is zero, the allocation and activities of this effort must follow suit in order for the model to function properly. The next two graphs below shows two important variables for testing. When the perceived need for testing is zero, we observe that the daily manpower allocated for testing is constantly zero through the entire software project cycle. When the daily manpower is zero, the testing rate is also zero as no workers are available to perform the testing effort.

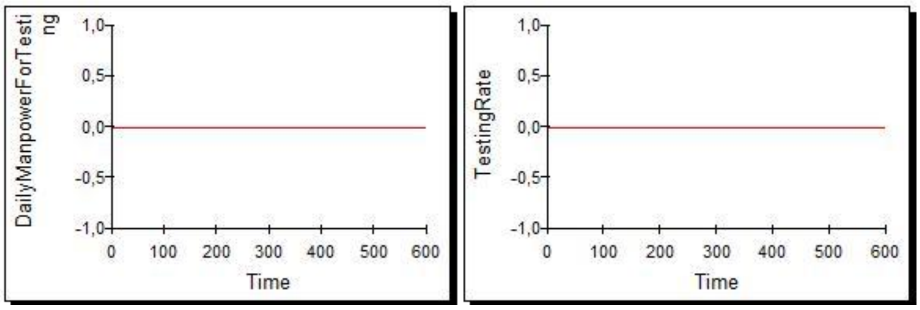


Figure 54. Daily manpower for testing and testing rate,
Under extreme condition

When no manpower is allocated for the testing effort, and no testing is carried out, the amount of active and passive errors that are detected in testing equals zero. From graphs below we see that through the simulation, not a single active or passive error is found. When manpower is not allocated, the testing effort shuts down completely. This leads to a subsequent halt in any results stemming from the testing effort. This is evident in the two graphs below.

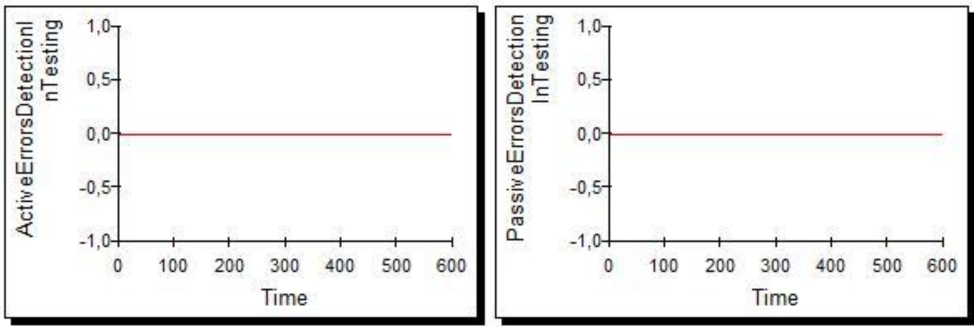


Figure 55. Active errors detection in testing and passive errors detection in testing, Under extreme condition

The graph depicted below highlights the complete shut-down of the testing effort. The graph shows the total number of detected errors in testing. From the two earlier graphs, the detection of active and passive errors were zero. In order for the model to follow the natural logic and provide true representations of reality, the last stock must also be equal to zero. This fact is evident from the graph below, as the total number of errors detected in testing is following suit. Not a single error is found by the testing effort.

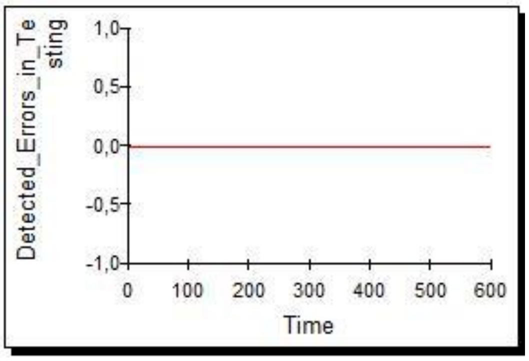


Figure 56. Stock of detected errors in testing under extreme condition

As we can see from my results, all related variables and stocks go to zero when no manpower is allocated to the testing efforts inside the production cycle. There is no erroneous testing carried out, and there are no negative stocks. There are no detected errors in testing that could have been fed back to production to get reworked. This is shown in the final graph below by the green line. According to the base model assumption the testing process will be activate at the end of coding phase when 80% of the tasks are done. That is why the detected errors waiting for rework under this extreme situation are fewer near to the end of the project since the testing process is not activated. The model reacts logically to the extreme conditions imposed and the robustness is strengthened.

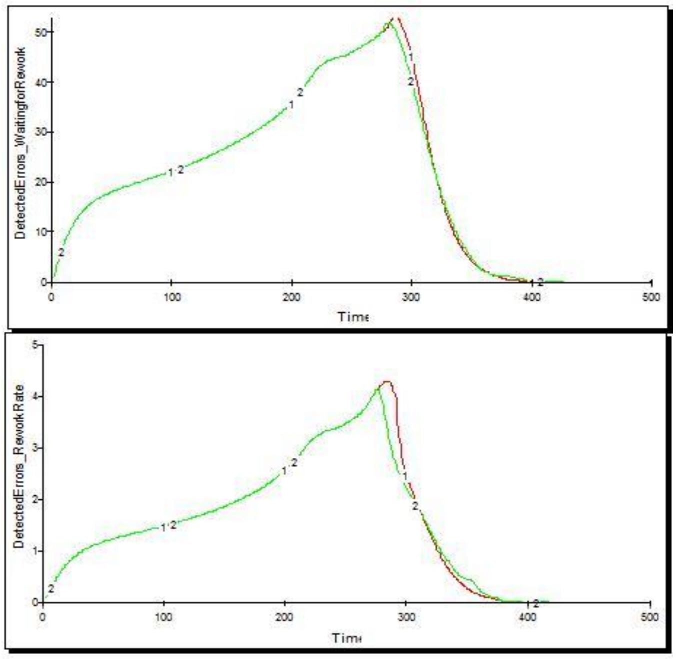
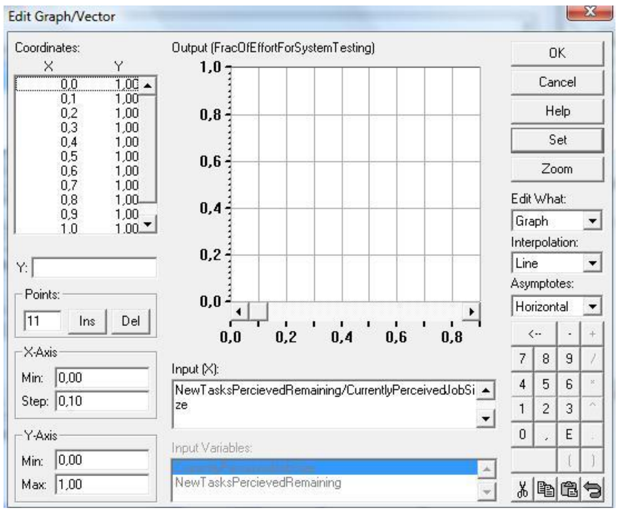


Figure 57. Detected errors waiting for rework and detected errors rework rate, Red line is under normal and green line is under extreme condition

The second extreme conditions test I will perform is linked to software development. The entire model rests on the development of software. No other effort can start unless tasks are being worked and completed. The rate of software being developed hinges on the daily manpower allocation for software development. In this test I will remove all daily manpower allocated to the software development effort. So the first step is to set the effort for development to zero, so that no manpower is allocated for this effort.

As we explained in the previous extreme condition, in this model the manpower-allocation for development and testing efforts hinges on a graph-function of the “Fraction of testing effort”. The first figure below shows the initialization of no manpower for development. The fraction of testing effort is now held at one, and all manpower is allocated to testing. The result is a situation where no manpower is allocated and the development process should not commence.


Edit Graph/Vector

Coordinates:

X	Y
0,0	1,00
0,1	1,00
0,2	1,00
0,3	1,00
0,4	1,00
0,5	1,00
0,6	1,00
0,7	1,00
0,8	1,00
0,9	1,00
1,0	1,00

Output (FracOfEffortForSystemTesting)

Points: 11

X-Axis Min: 0,00 Step: 0,10

Y-Axis Min: 0,00 Max: 1,00

Input (X): NewTasksPercievedRemaining/CurrentlyPerceivedJobSize

Input Variables: NewTasksPercievedRemaining

Edit What: Graph

Interpolation: Line

Asymptotes: Horizontal

Figure 58. Fraction of effort for system testing

This should lead to a situation where no tasks are being worked or accomplished. This in turn collapses the entire system, as the other efforts have manpower, but no tasks to investigate, test or rework. All stocks and rates in the system should go to zero, and no activity will be present at any stage. If there is any activity going on despite the fact that no tasks are completed, there are some problems with the model that I must address. The figure below shows the stocks and variables under this condition.

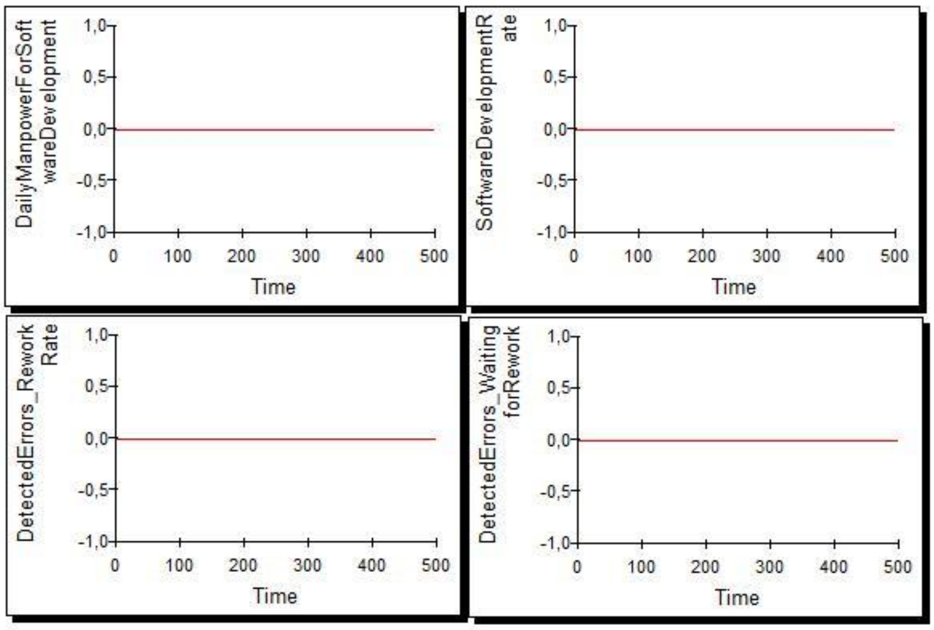


Figure 59. The stock and variables under this extreme condition

7.2.4 Recreating the reference mode

In chapter five I presented a reference mode for this model. The reference mode is a set of graphs or other descriptive statistics that explains the problem at hand. The system dynamic model is created to simulate a problem, before it can provide a solution. It is therefore vital that my model with the enhancements can simulate the key challenges of the software crisis. If the model is unable to replicate this behavior, it holds little validity. In chapter five I gave a detailed introduction to the reference mode, and the problem at hand. Based on this explanation the reference mode here are the detected errors waiting for rework, and the detected errors rework rate. These two graphs will now be presented below, where the first simulation-run is the base-model whiles the second run is the enhanced model.

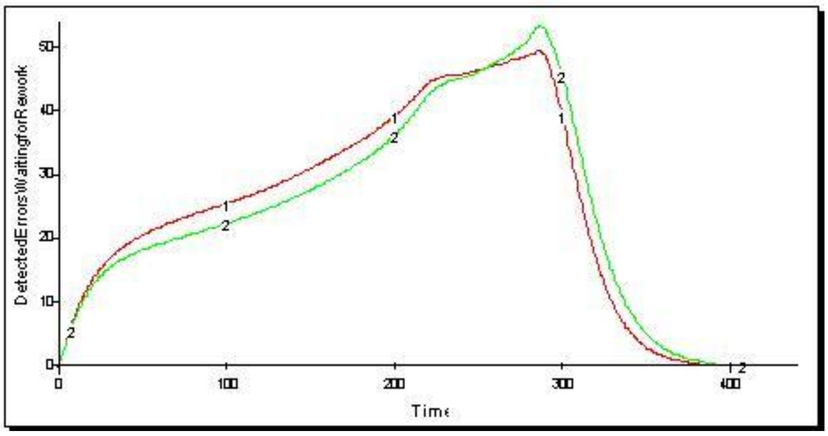


Figure 60. The stock of detected errors waiting for rework
Red line is base model and green line is enhanced model

The first graph shows the accumulation of detected errors waiting for rework. The software project is underway, and the quality assurance effort starts to detect errors. These errors are sent for rework in order to be corrected. At the beginning we see the increase in detected errors waiting to be reworked in both graphs. After some time, the rework-effort has initialized, and the increase in errors waiting to be reworked decreases as they are corrected. In my model this decrease is more profound, but the behaviors of the two graphs are the same. The reduction in increase is linked to learning as the project progress and moves from design to coding, and the effort needed per error due to error type. Coding-errors are easier to rework than design errors, although they have a higher magnitude. Hence the increase in detected errors is still present. There is another factor contributing to the reduction in increase my model-simulation. The capture re-capture system that has been integrated allocates part of quality assurance to the tagging effort. When errors are detected and get reworked, some of quality assurance staff will be allocated to tag releasing activities to tag the fixed errors. When parts of the staff are busy with tagging, the manpower available for error detection is slightly lower. The detected errors waiting for rework is influenced by this and therefore is lower.

The number of detected errors still waiting to be reworked starts to increase steadily as the project moves along. The reason for this behavior rests on several factors. When production progress and moves from design to coding, the productivity due to learning also increases and more tasks are completed. The share number of tasks that is completed increases the numbers of tasks containing errors. Additionally, the relatively easy effort to rework coding-errors leads to motivational losses. Rework takes a programmer away from creativity and over to maintenance. This leads to losses of interest and the rework-rate lowers.

Schedule pressure along with workforce mix is also influencing the rework rate, and results in a peak of errors waiting to be reworked before the project reaches the end of the production cycle. In my model this peak is more aggressive than in the base-model run, but the behavior is similar. The more aggressive peak at the end of my model is also due to feedback from testing. Since the testing effort sends errors back to rework when they are identified as bad fixes, the amount of detected errors waiting for rework increases. Secondly untagged errors in testing are sent back to quality assurance for identification, and also end up for rework as soon as the error has been identified. After the end of the production no more tasks are completed. This equals to no more errors are detected and waiting for rework, and the stock drops to zero as the rework team finishes their job. The slope is convex as workers are moved from rework to testing during the final days of rework.

From the graph it is evident that my enhanced model reproduces the reference mode in regards to the stock of detected errors waiting for rework.

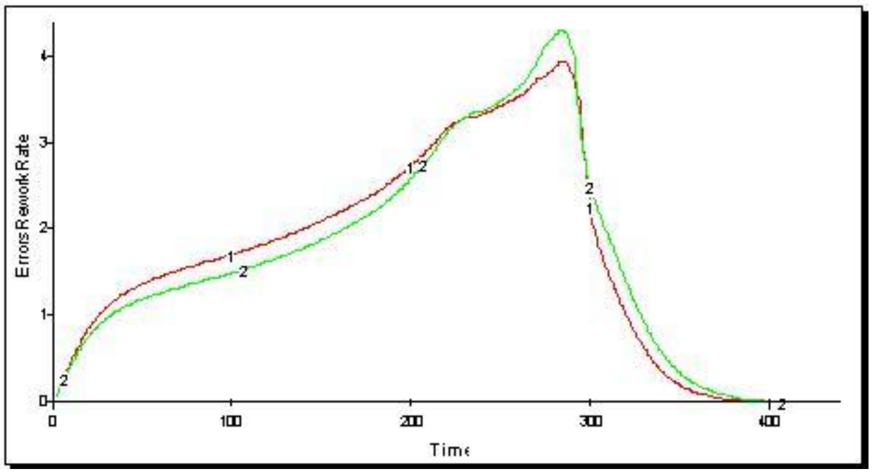


Figure 61. Detected errors rework rate,
Red line is base model and green line is enhanced model

The second graph that is included in my reference mode is the detected errors rework rate. In chapter 5 I provided a detail explanation on how the rework-rate was influenced by software productivity, motivation, the errors type, and manpower allocation. As soon as errors are detected and need to be reworked at the early stage of the project, the rework team is allocated to initialize the correction-process. From the graph we see how in both cases the rework-rate increases as errors are being corrected. The rate of correction starts out strong in both cases as there is an aura of excitement around the new project. Therefore the initial effort put into the earlier tasks are higher as motivation and optimism is high.

The initial boost of increase is not maintained however. The reason for this lies in the effort and the exhaustion level. In the early stages of development the project is located in the design phase. Design-errors are harder to rework than coding errors, and the exhaustion-level increases. When workers starts to feel exhausted, they become less productive. Hence the rework-rate decreases. This is observed for both graphs as the initial boost of rework the rework rate starts to level off. In my enhanced model the effect is somewhat larger, and the green line follows a shallower trajectory.

Rework-teams are also allocated from development, meaning that every worker allocated to rework is unable participate in the development of the software. This leads to negative associations with rework, and the effort could be regarded as punishment. This lowers productivity and provides motivational losses Fatigue equals less tasks being completed, and less errors detected by quality assurance. The result is a lower rework rate. The third factor is manpower-allocation, as management relocates manpower from rework to production if possible. With less errors waiting to be reworked, less manpower is needed and staff is reallocated. The fourth reason for the decline that is exclusive to my model is the captures re-capture effort. In the graph we observe that my decline in effort is higher in magnitude than the base-model. The reason for this rests in the phenomenon explained earlier in this chapter. The capture re-capture effort allocates some parts of the quality assurance team to tagging errors. This reduction in manpower leads to a lower rate of error detection. Therefore the managerial decision reduce rework manpower based on detected errors waiting for rework since there is less detected errors waiting for rework. Hence, my green line is following the pattern of the red base-line but at a lower numerical value.

These factors lead to the slower increase in the rework rate for a period of time. However, as productivity picks up again, and the exhaustion-levels drop, the rework teams are increasing their productivity as well. The effect of learning as the project moves from design to coding, and the small effort needed per coding error increases the rework-rate as well. The workforce mix leans towards more experienced workers, and at this stage rework rate increase. Due to the same reasons, development rate of tasks increases as well as the generation of errors. This increases the amount of detected errors waiting for rework, and management allocates more manpower to rework. The rework efficiency goes into its sharpest increase since the initialization of the project. Since my model held a shallower trajectory at the first level-off in the rework-rate, the effect of these factors is higher. Both graphs show an increase in the rework-rate, but my model’s peak is higher than the base-model. However, the difference is numerical and not behavioral. Hence the shape of the graphs remains similar. This is due to detected errors in testing phase that are fed back from testing to rework at the end of the production phase. This increases the detected errors waiting for rework to a higher degree than in the base model. Management allocates more manpower to rework as a response peak for my model.

In the end the tasks have been completed, and the software production effort grinds to a halt. The rework-effort is still occupied with tasks waiting to be reworked, but is slowly phasing out. Both lines show a convex curve, as the number of tasks waiting for rework drops. During these final stages of rework, management allocates people from rework to testing. Hence the line levels off near the end of the project as it reaches zero.

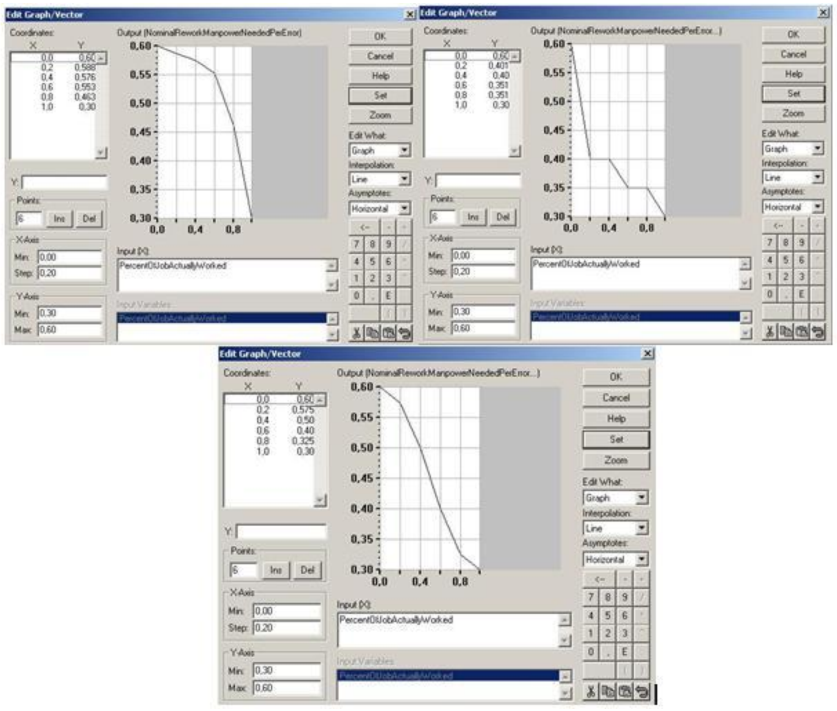
In this test I have provided one important stock and one important variable for the reference mode that highlights the issue of software quality and rework. The results reveal that even though my numerical values are different than the base-model the effects and behaviors are the same. This strengthens my enhanced model's validity as it is able to reproduce the reference mode to a satisfactory degree.

7.2.5 Testing Table functions

The next test applied in my paper is the table function test. This test focuses on the table-functions used in a model, and provides a justification for the influence of these functions on the modeled behavior. In every table function we make a parameter assumption based on empirical data, or verbal description of a phenomenon. These assumptions must be tested, in order to strengthen their validity. It is of vital importance that the behavior observed in the model stems from the feedback-loops and causal relations inside the model. Hence, the distribution between two determined parameters in a table function should only change the numerical values of the data-points. If the loops and causal relationships are the determining factor, the behavior of the system should be unaltered by this change in distribution. If the behavior is altered significantly, it is highly sensitive to the parameter assumption. This indicates that an un-modeled structure is causing the effect on the system, and the table-function is not applicable. The model needs to be expanded with a new structure to accommodate this result.

In the figures below I provide three graphs that show three different distributions between the documented parameters. I will provide a brief explanation of the logic behind the shape of distribution, and the graph that is chosen for the model. There are in total four tests carried out.

The effect of Errors type on rework effort needed per error:


Edit Graph/Vector

Output (NominalReworkManpowerNeededPerError)

Coordinates:

X	Y
0,0	0,60
0,2	0,588
0,4	0,576
0,6	0,553
0,8	0,463
1,0	0,30

Points: 6

X-Axis Min: 0,00 Step: 0,20

Y-Axis Min: 0,30 Max: 0,60

Input (X): PercentOfJobActuallyWorked

Edit What: Graph
Interpolation: Line
Asymptotes: Horizontal

Edit Graph/Vector

Output (NominalReworkManpowerNeededPerError...)

Coordinates:

X	Y
0,0	0,60
0,2	0,401
0,4	0,40
0,6	0,351
0,8	0,351
1,0	0,30

Points: 6

X-Axis Min: 0,00 Step: 0,20

Y-Axis Min: 0,30 Max: 0,60

Input (X): PercentOfJobActuallyWorked

Edit What: Graph
Interpolation: Line
Asymptotes: Horizontal

Edit Graph/Vector

Output (NominalReworkManpowerNeededPerError...)

Coordinates:

X	Y
0,0	0,60
0,2	0,575
0,4	0,50
0,6	0,40
0,8	0,325
1,0	0,30

Points: 6

X-Axis Min: 0,00 Step: 0,20

Y-Axis Min: 0,30 Max: 0,60

Input (X): PercentOfJobActuallyWorked

Edit What: Graph
Interpolation: Line
Asymptotes: Horizontal

Figure 62. The effect of Errors type on rework effort needed per error

The first table function I analyse in this test is the effect of error type on manpower allocation for rework activities. From the base-model we learn that the error type influences the effort needed per error. In the first half of a software project, the production is in the design stage. Programmers are writing the core design-codes that constitute the fabric and blueprint of the software. If there is an error in these codes, they have a significant effect on the software's performance. These errors require attention and concentration to rework properly, and the effort to do so is high. In the second half of the project the production enters the coding stage. In this stage the codes written are lighter application and function-codes. They are primarily codes that operate the system within the design and blueprint. Code-errors are easier to rework, and the effort needed per error less.

Additionally, during the software progress workers learn how the system should operate. This knowledge allows them to understand an error quicker, and how it should be reworked. This is explained in greater detail in chapter 6 under the model structure enhancement for rework-manpower needed per average error.

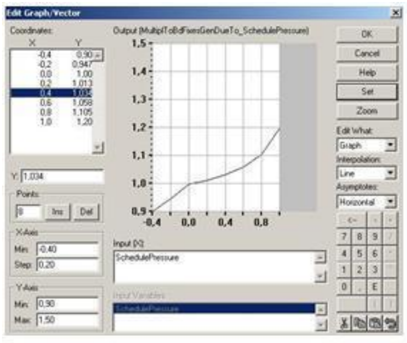
In this test, we have a multiplier that runs from 0.6 to 0.3. This parameter is chosen based on the empirical data and explanation provided by Abdel-Hamid in the original model (Abdel-Hamid, 1991: p.107). In the different tables above I present three different scenarios of distribution to the rework effort needed per error. In the first graph we have a situation where the rework effort needed per error due to errors type changes slowly between design and coding errors until it holds a dramatic effect into the project. This would indicate that the design errors of any kind are very hard to rework, and that the learning-effect of progress is minimal at first. Note that the effect carries into the early coding stage as well, and that fundamental code is also hard to rework.

The second scenario depicted by the figure is a situation where the effects of errors type and learning hold a step-wise influence on the rework effort. Initially there is a step drop in rework effort needed per error in the middle of the design phase. Workers learn very quick how to rework design errors, and the effort needed for errors committed found after progressing from fundamental design to more rudimentary design-tasks are significantly smaller. However, the effect levels completely off for a period of time. The design-errors are equally hard to rework until the project reaches the coding stage. This means in effect that learning is not taking place between middle-part of design and the beginning of the coding phase. The same pattern is repeated as the effect on effort for rework drops considerably from the first coding-errors to the middle of the effort. Then the effort levels off again, and is kept stable until the final approach of coding effort. Then the level is lowered to 0.3 at the very end of the project.

The third and final scenario depicts a smoother distribution between the different stages. The effect of errors type and learning between the design stage and the coding stage holds a continuous effect through the progress from design to coding. We observe that the effort needed per design error is decreasing continuously towards the coding stage, and that this behaviour carries over into that production stage. This follows the logic of continuous learning during rework, and the nature of the errors types. Design-errors are initially harder to rework, but as they progress from fundamental design to rudimentary design the difficulty level drops incrementally. Secondly, the first coding errors are indeed easier to rework than design errors but harder to rework than the last coding errors for the same reason.

The result of this test shows that the different tables do not change the behaviour of the model. The numerical values do show variation, but the behaviour is similar. Hence, we may utilize a graph-function here. The function that we choose here is the third distribution. This distribution is based on both empirical description of effort needed for rework due to errors type. Secondly the numerical values are provided by empirical studies on this subject (Abdel-Hamid, 1991: p.107).

Effect of Schedule pressure on bad fixes generation:

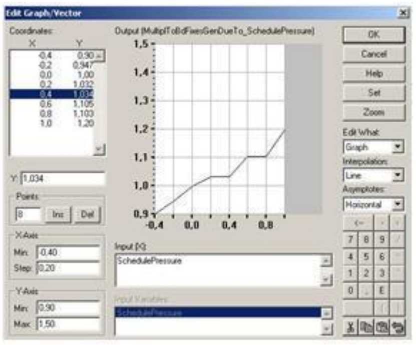


Edit Graph/Vector

Coordinates:

X	Y
-0.4	0.90
-0.2	0.947
0.0	1.00
0.2	1.013
0.4	1.034
0.6	1.058
0.8	1.105
1.0	1.20

Y: 1,034
Points: 8
X-Axis Min: -0.40 Step: 0.20
Y-Axis Min: 0.90 Max: 1.50
Output (MultplToBdFixesGenDueTo_SchedulePressure)
Input (X): SchedulePressure
Edit What: Graph
Interpolation: Line
Asymptotes: Horizontal

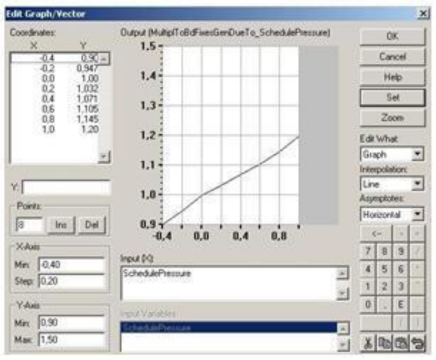


Edit Graph/Vector

Coordinates:

X	Y
-0.4	0.90
-0.2	0.947
0.0	1.00
0.2	1.032
0.4	1.034
0.6	1.105
0.8	1.103
1.0	1.20

Y: 1,034
Points: 8
X-Axis Min: -0.40 Step: 0.20
Y-Axis Min: 0.90 Max: 1.50
Output (MultplToBdFixesGenDueTo_SchedulePressure)
Input (X): SchedulePressure
Edit What: Graph
Interpolation: Line
Asymptotes: Horizontal



Edit Graph/Vector

Coordinates:

X	Y
-0.4	0.90
-0.2	0.947
0.0	1.00
0.2	1.032
0.4	1.071
0.6	1.105
0.8	1.145
1.0	1.20

Points: 8
X-Axis Min: -0.40 Step: 0.20
Y-Axis Min: 0.90 Max: 1.50
Output (MultplToBdFixesGenDueTo_SchedulePressure)
Input (X): SchedulePressure
Edit What: Graph
Interpolation: Line
Asymptotes: Horizontal

Figure 63. Effect of Schedule pressure on bad fixes generation

The second test of table functions I present concerns the effect of schedule pressure on the generation of bad fixes. Bad fixes occur when an attempt to fix an error is done incorrectly. In chapter six I explained how stress and schedule pressures leads to an increase in the bad fixes committed. When a software program run behind schedule or the completion-day is imminent, the stress and pressure on the workforce increases. This leads to a situation where stress leads to poor judgment, lack of focus, and errors made due to haste. For a more detailed explanation see chapter 6 under the model structure enhancement for bad fixes generation.

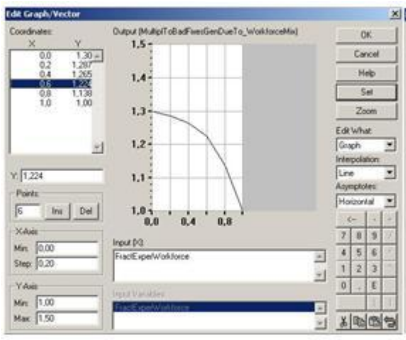
In the first graph depicted, the influence of schedule pressure is varying over time. Under negative schedule pressure, in the initial increase in schedule pressure the factor on bad fixes increases steadily from 0 to 1. Then under positive schedule pressure, the effect is reduced in strength for a period of time. Therefore, the increase in bad fixes is following a smoother increase-pattern towards the end of the project. Right before the end of the project the effect increases again to the same level seen in the initial effect and it peaks towards 1.2.

In the second graph the effect of schedule pressure on bad fixes follow a step-wise pattern. The effects of schedule pressure only produce an increase in bad fixes for a certain time period. In between the intervals the level of bad fixes are kept at the same rate. This would assume that the effects of stress do initially heighten the rate of bad fixes, and that this effect keeps the level of bad fixes constant for a period of time. More schedule pressure then bumps the effect of bad fixes upwards again, and the graph is given a stair-case behavior. Initially the schedule pressure leads to a smooth increase in bad fixes, but then the effect is more jagged.

The last graph shows an increase in bad fixes due to schedule pressure that is continuously increasing. Stress factors are well documented in the field of employment. Stress and occurs in every form of work-environment, and leads to problems in quality. This factor is described by Wickens and Holland as "…When under stress, people usually persist in trying the same erroneous solutions to a problem. This leads to an additional cause of stress" (Wickens & Holland: 2000). In this last graph the effect of schedule pressures on bad fixes is continuous and cumulative. Hence, the behavior of the graph is a straight increasing line between 0.0 for no schedule pressure and 1.2 for heavy schedule pressure.

The simulations for these graphs show the same behavior for all three versions. The model's behavior is not affected by the distribution between 0 and 1.2, and therefore it is valid to have a table-function for this relationship. The third graph provides the correct behavior of the phenomenon based on the empirical data available by for example Jones and Bonsignour (2012). Secondly, this behavior fits the empirical descriptions made of the effects of stress on errors in a work-environment.

Effect of workforce mix on bad fixes generation:



Edit Graph/Vector

Output (MultiplToBadFixesGenDueTo_WorkforceMix)

Coordinates:

X	Y
0,0	1,30
0,2	1,287
0,4	1,265
0,6	1,224
0,8	1,138
1,0	1,00

Y: 1,224

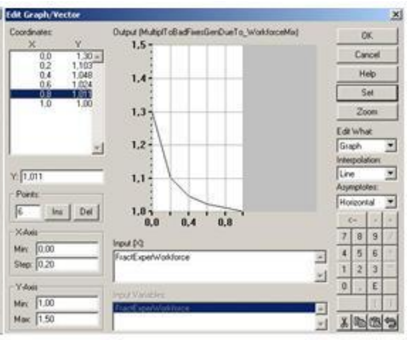
Points: 6

X-Axis Min: 0,00 Step: 0,20

Y-Axis Min: 1,00 Max: 1,50

Input (X): FractExpeWorkforce

Edit What: Graph
Interpolation: Line
Asymptotes: Horizontal



Edit Graph/Vector

Output (MultiplToBadFixesGenDueTo_WorkforceMix)

Coordinates:

X	Y
0,0	1,30
0,2	1,103
0,4	1,048
0,6	1,024
0,8	1,011
1,0	1,00

Y: 1,011

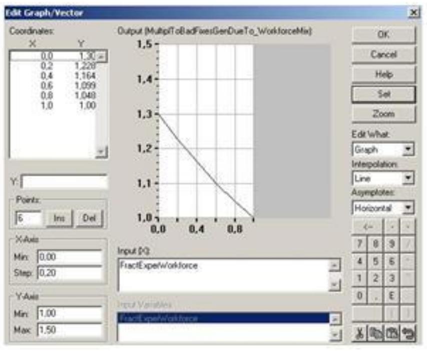
Points: 6

X-Axis Min: 0,00 Step: 0,20

Y-Axis Min: 1,00 Max: 1,50

Input (X): FractExpeWorkforce

Edit What: Graph
Interpolation: Line
Asymptotes: Horizontal



Edit Graph/Vector

Output (MultiplToBadFixesGenDueTo_WorkforceMix)

Coordinates:

X	Y
0,0	1,30
0,2	1,228
0,4	1,164
0,6	1,099
0,8	1,048
1,0	1,00

Points: 6

X-Axis Min: 0,00 Step: 0,20

Y-Axis Min: 1,00 Max: 1,50

Input (X): FractExpeWorkforce

Edit What: Graph
Interpolation: Line
Asymptotes: Horizontal

Figure 64. Effect of workforce mix on bad fixes generation

The effect of work-force mix on bad fixes is the third table-function test in my thesis. The amount of experienced workers in the workforce influences the error-proneness of the staff. Experienced workers have participated in rework-efforts earlier, and have valuable knowledge of the software that is being produced. New workers must acquire this knowledge during the production-stages and their inexperience leads them to make more mistakes. Therefore the workforce-mix on the generation of bad fixes is important. The parameter for the table is between 1 and 1.3. For more details see chapter 6 under the model structure enhancement for bad fixes generation.

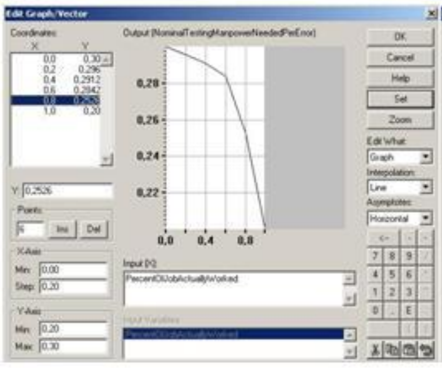
In the first graph we have a scenario where initially the move from only inexperienced workers to a mix of 50/50 between inexperienced and experienced, yields little effect on the generation of errors. From this point on however, the increase in workforce-mix towards fully experienced staff increases sharply.

In the second graph of bad errors due to workforce mix the scenario is opposite. The initial move from total inexperienced staff to a semi-experienced staff yield a sharp decline in bad fixes. The effect of experience is decreasing over time, and is clearly lower as the workforce mix approaches a fully experienced staff.

The third graph depicts a scenario where the effect of work-force mix is continuous. For every worker that is experienced, the bad-fix generation rate decreases. The decrease is a straight line, as more experienced workers bring less bad fixes. This follows the logic of learning and progress. Experienced workers have both experience from previous work, and the capacity to learn about the current project faster. Learning is a continuous process, and the effect on bad fixes increases constantly.

The three graphs provide the same behavior for the model when simulated. It is not sensitive to changes in distribution and the validity is strengthened. The graph that I use in this model is the third option, as it follows the logic and parameters provided by the base model of Abdel-Hamid, and the descriptive data of Jones and Bonsignour. This is explained at greater detail in chapter six. The effect of experience is continuous, and it is a valuable source for management to ensure software quality and avoid bad fixes (Abdel-Hamid, 1991: p.108) (Jones and Bonsignour, 2012).

The effect of Errors type on testing effort needed per error:



Edit Graph/Vector

Output (NominalTestingManpowerNeededPerError)

X	Y
0.0	0.30
0.2	0.296
0.4	0.2912
0.6	0.2842
0.8	0.2526
1.0	0.20

Y: 0.2526

Points: 6

X-Axis Min: 0.00 Step: 0.20

Y-Axis Min: 0.20 Max: 0.30

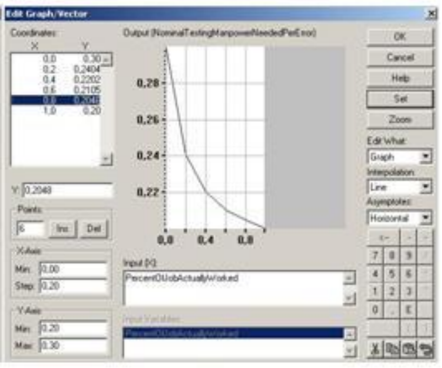
Input (X): PercentOfJobActuallyWorked

Input Variables: PercentOfJobActuallyWorked

Edit What: Graph

Interpolation: Line

Asymptotes: Horizontal



Edit Graph/Vector

Output (NominalTestingManpowerNeededPerError)

X	Y
0.0	0.30
0.2	0.2404
0.4	0.2202
0.6	0.2105
0.8	0.2048
1.0	0.20

Y: 0.2048

Points: 6

X-Axis Min: 0.00 Step: 0.20

Y-Axis Min: 0.20 Max: 0.30

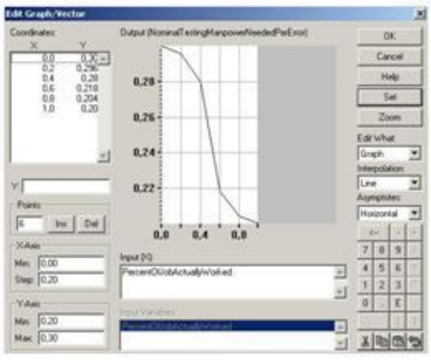
Input (X): PercentOfJobActuallyWorked

Input Variables: PercentOfJobActuallyWorked

Edit What: Graph

Interpolation: Line

Asymptotes: Horizontal



Edit Graph/Vector

Output (NominalTestingManpowerNeededPerError)

X	Y
0.0	0.30
0.2	0.296
0.4	0.28
0.6	0.218
0.8	0.204
1.0	0.20

Points: 6

X-Axis Min: 0.00 Step: 0.20

Y-Axis Min: 0.20 Max: 0.30

Input (X): PercentOfJobActuallyWorked

Input Variables: PercentOfJobActuallyWorked

Edit What: Graph

Interpolation: Line

Asymptotes: Horizontal

Figure 65. The effect of Errors type on testing effort needed per error

The fourth and final table-function that is validated in this chapter is the effect of error-type on the testing effort. From the first table function we learned that the type of error holds an effect on the effort needed to rework them. The same logic applies to the testing effort as well. Design errors are active errors; hence they are making the testing effort more challenging. Active errors multiply over time, as they cause new errors to appear at a lower level. One active error can alone cause ten-folds of passive errors. Code-errors are all passive errors causing no additional errors by their existence.

The parameter here is set between 0.3 and 0.2, by the logic explained in chapter 6. In order to calculate a multiplier I utilized the logic for the rework effort multiplier due to error type. In the original model this multiplier held a 0.6 for design, and 0.3 for code-errors. In this case however, the testing team is investigating the errors not reworking it. When an application is ready, the testing team runs it to test its functionality. Then if there is an error, it is classified as either rework-considerable or no rework-considerable. For the rework-considerable deviations, the application is dismantled and the code-structure investigated. Here the testing-team identifies the error as caused by either tagged or untagged pieces of code. The effort holds different steps, but is still lighter in effort than reworking errors. Hence the effort is assumed to be 0.3 for the design errors. Coding errors are given a value of 0.2 since they are by default easier to classify but their magnitude is high.

The first scenario provides a situation where the effect on effort is significantly different between the design-errors and coding errors. Initially there is little reduction in effort while moving from the design-phase to the coding-phase. Effort is decreasing, but kept high through the entire design-phase. When the project moves into coding, the effect on testing-effort is significant. The effort drops sharply from each coding-stage until the multiplier reaches 0.2. This gives the multiplier a more concave decreasing line, where the effect is significant towards the end of production for the effort needed in testing. In this scenario, the effect of learning in the testing effort is at a minimum initially, but maximum at the end.

The second scenario is opposite from the example above. Here, the initial progress from fundamental design towards more rudimentary design-tasks holds a significant effect on the testing effort. The line is a more convex distribution between the values of 0.3 and 0.2. In this scenario learning is initially valuable as the testing of design-errors provide significant decrease in the effort needed. Secondly, the effort level is very different between different kinds of design errors. This effect levels off as the project is moving towards the coding phase, where the effort falls to the 0.2 level

The last scenario is an s-shaped distribution between the two values. Initially as the project moves from fundamental design to rudimentary design, the effort is reduced at a slow rate. The staff is learning about the software during testing, but the design-errors are still causing high effort. Through the testing-stages more error-patterns are revealed for the testing-teams, and their knowledge of the patterns reduces effort between every stage. This effect, together with the relative decrease in difficulty as the project moves from design to coding, provides a steeper decrease in the effort needed due to error type. This steep decrease continues as the project moves from design to coding. However, after a period of time the effect levels off towards 0.2. In the end the coding-errors are simple to test, classify and identify, and the effect of learning does not hold the same impact.

For the fourth and final test, the result remains the same as for the previous three. The model is not sensitive to the distribution between the values of the parameter. The behavior of the model remains the same, while the numerical values are different between each scenario. Therefore I can with more certainty assume that the table-function is appropriate and valid. The table that is chosen for the final test is the third option. The logic behind the effort is provided by the same logic as for the rework effort. Error-type has an influence on the effort needed for testing. This effort decreases as the project moves from the design-stage to the coding stage. Initially the effort-reduction is slighter between the first stages of design-testing, but drops significantly as the project moves towards coding due to the nature of the error type and learning. The effect diminishes as the coding-errors are easier to test, classify and identify, but their magnitude is high so it’s a gradual decrease. This would provide an S-shaped graph-line. The parameter between 0.3 and 0.2 is calculated based on the base model of Abdel-Hamid (Abdel-Hamid, 1991: pp.112-113) and the empirical data by Jones and Bonsignour. (Jones & Bonsignour, 2012: pp.338-340). More details are available in chapter six.

This concludes my table-function test. The test reveals that my model is not sensitive to the table functions. This strengthens my model, as no un-modeled structure is found to influence the behavior of the model when I have included my enhancements.

7.2.6 Loop-knockout analysis

The cutting loop test or loop-knockout analysis is a method to search for behavior anomalies in the model, and establish the strength and significance of loops. Sterman describes the procedure as a common method where one zero out the target relationship. This will effectively “cut” the loop and it holds no effect. This is done by for example setting adjustment-time for a stock to an indefinite amount. For example, if the hiring-policy of a company is “Available applicants”/”Adjustment Time” setting an indefinite adjustment-time would create a situation where nobody gets hired. A second form of loop cutting is done by setting an information delay to an infinite amount. It is also possible to cut loops by setting non-linear functions y=f(x) to unity of all values of x. The result is a knocked out loop that yields no effect on the behavior of the model. In short, it negates all influence as if the loop did not exist (Sterman, 2003: p.880).

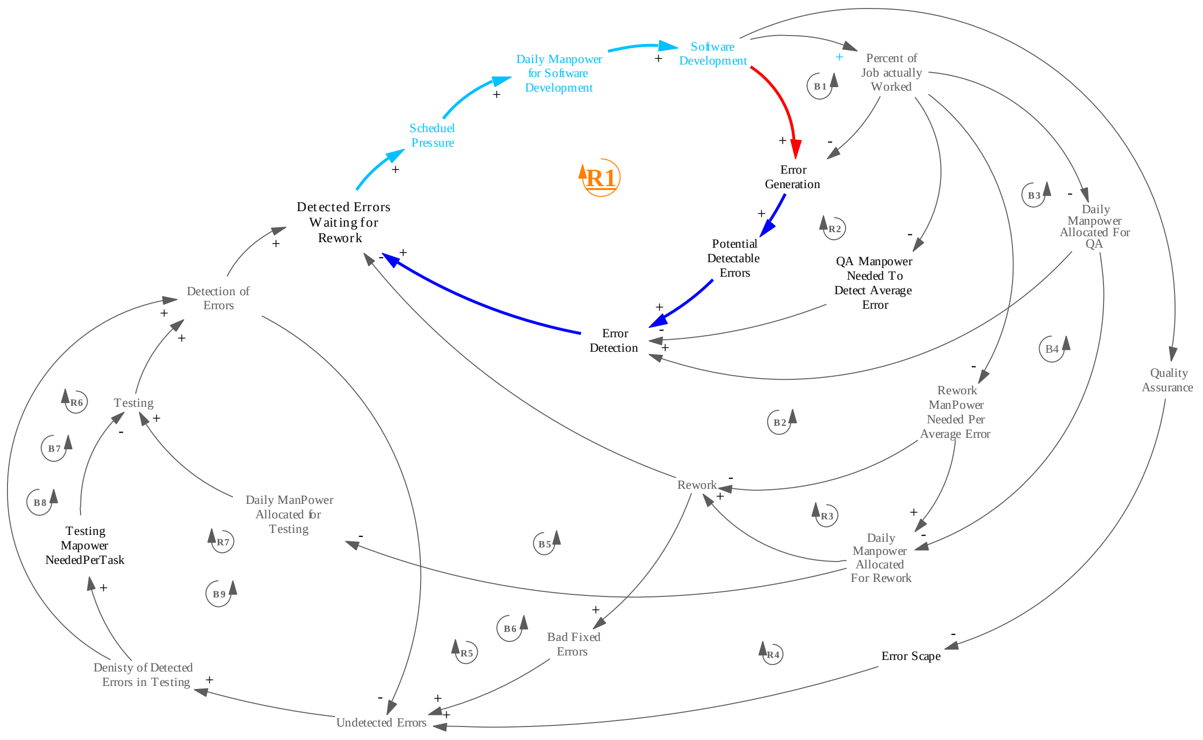
In this test I will cut a loop in order to see the strength and influence of the loop on my reference behavior. The procedure is to cut a loop by removing the effect of one variable on another variable in the loop. This will cause the chain of relationships between the variables to seize. After a loop has been knocked out, I look at the behavior of the variable at the cut—point and the behavior of the reference behavior before and after cutting. In this test I will provide graphs from the reference variables. The lines marked as 1 are the simulations before cut, while graphs designated as 2 are after cut. In this way we see the effect of the loop and if anomalous behavior occurs. If the loop changes the behavior in the reference variables it is a major loop. If the changes are minor, it is a minor loop.

I would also like to make a note of a difference in the procedure here due to the nature of the model. Since the variables are intertwined, it is hard to cut only one loop at the time. For cut number four to eight, I cut multiple loops simultaneously. In this fashion I can compare the results, and determine which of the loops hold strongest influence based on the observed behavior.

In this test I will first present a big CLD of the subsectors in interest, namely the quality assurance and rework and testing. The loop that is cut is highlighted as the marker is given in a bigger font. The red line indicates where the loop has been cut. Then I proceed by presenting three graphs that show the behavior before and after cutting, and I comment on the behavior.

7.2.6.1 Cutting loop R1

The first loop I choose to cut is the first reinforcing loop of the system. The loop governs the relationship between allocation of manpower for software development, software development, errors generation and errors detection activity. This is a reinforcing loop that holds an important influence on the detected errors waiting for rework through the error-generation rate due to the increase in the total amount of errors, as more tasks are being accomplished. The effect should be significant. The loop is cut by setting the software development rate to a constant with the initial variable value.



Software Development
Daily Manpower for Software Development
Scheduel Pressure
Percent of Job actually Worked
B1
R1
Error Generation
Detected Errors Waiting for Rework
Potential Detectable Errors
R2
QA Manpower Needed To Detect Average Error
B3
Daily Manpower Allocated For QA
Detection of Errors
Error Detection
B4
Quality Assurance
R6
Testing
B7
B2
Rework ManPower Needed Per Average Error
B8
Testing Mapower NeededPerTask
Daily ManPower Allocated for Testing
R7
Rework
R3
B5
Daily Manpower Allocated For Rework
B9
B6
Bad Fixed Errors
R5
R4
Error Scape
Denisty of Detected Errors in Testing
Undetected Errors

Figure 66. Quality assurance and rework and testing subsectors CLD, with cutting loop R1

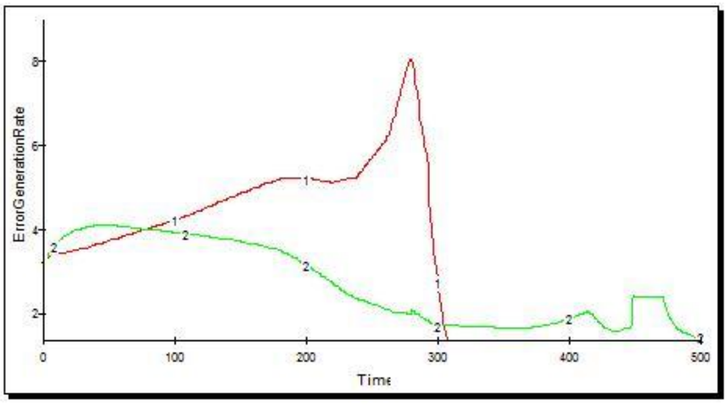


Figure 67. Errors generation rate ,
Red line is before and green line is after cutting loopR1

The first graph that is presented here is the error generation rate. Straigh from the start we see that cutting this loop causes a big change in the variable's behavior. This is an anomolus behavior-pattern described by Sterman, signalling the significance of the loop. Errors are generated at a much lower level, and are continuing to decrease. However, the projct runs much longer than in the situation when Loop R1 is in effect. This is consistent with cutting the progress of software development to a constant rate with no dynamic effects of productivity,schedule pressure and workforce mix. As a result tasks are accomplished at a much slower rate. Therefore with less tasks developed, less errors generated.

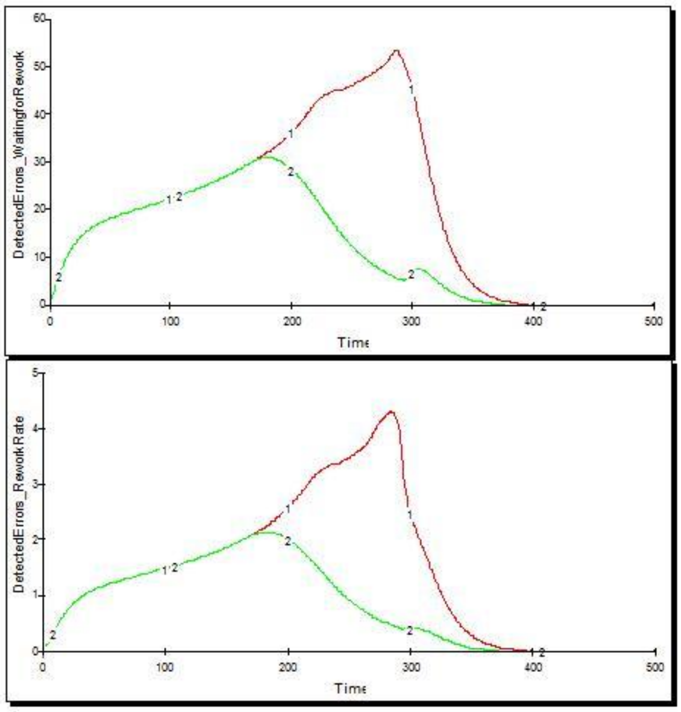


Figure 68. Detected errors waiting for rework and detected errors rework rate,
Red line is before and green line is after cutting loopR1

The two next graphs that are shown are the detected errors waiting for rework, and the rework rate. By cutting the loop we see that cutting progress of software development to a constant rate with no dynamic effects, leads to a situation where tasks are accomplished at a much slower rate. Additionally, less errors are generated as well. This causes the reduction in the amount of potential detectable errors and likewise the number of errors that are detected and waiting for rework. The numbers of errors waiting for rework declines steadily through the progress of the project into the coding stage.

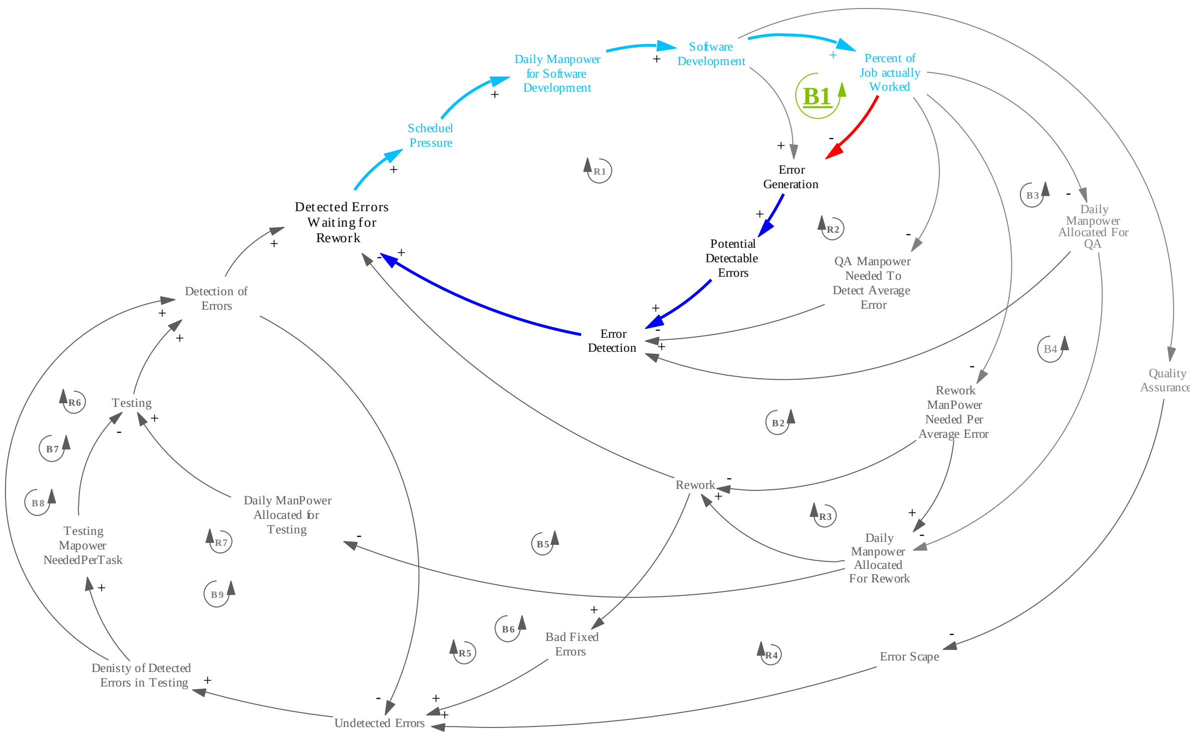
The detected errors rework rate follows suit, as the decreasing amount of detected errors that waits for rework, leads to managerial decisions to allocate less manpower to the rework effort. Another fact is that managers start to reallocate more manpower from quality assurance and rework to software development due to the lack of progress. Since development progress is set to a constant value however, these quality assurance and rework efforts declines through the entire coding phase. This increases the decline in the error detection rate and rework of errors decreases.

The effort is highly linked to the effects of productivity, schedule pressure and workforce mix and after cutting the loop these effects are removed. The difference in behavior shows this clearly. The cutting of loop R1 provides anomalous behavior, and changes the behavior of the reference variables. It is a major loop, and the validity of keeping it in the model is strong.

7.2.6.2-Cutting loop B2

The second loop for my loop-knockout analysis is the B1 loop. This loop balance the errors generation rate through progress of the project and the percent of job actually worked. The loop governs the relationship between allocation of manpower for software development, software development, errors generation and errors detection activity through the progress of the project, difference in errors type and productivity due to learning.

By cutting this loop we should observe the opposite behavior from the previous cut where loop R1 was involved. This loop is cut by replacing the variable of “Nominal Number of errors committed per task” to a constant with the initial variable value. The CLD shows the cut-point.



Daily Manpower for Software Development
Software Development
Percent of Job actually Worked
B1
Scheduel Pressure
R1
Error Generation
Detected Errors Waiting for Rework
Potential Detectable Errors
R2
B3
Daily Manpower Allocated For QA
QA Manpower Needed To Detect Average Error
Detection of Errors
Error Detection
B4
Quality Assurance
R6
Testing
B2
Rework ManPower Needed Per Average Error
B7
B8
Rework
Daily ManPower Allocated for Testing
R3
Testing Mapower NeededPerTask
R7
B5
Daily Manpower Allocated For Rework
B9
B6
Bad Fixed Errors
R5
R4
Error Scape
Denisty of Detected Errors in Testing
Undetected Errors

Figure 69. Quality assurance and rework and testing subsectors CLD, with cutting loop B1

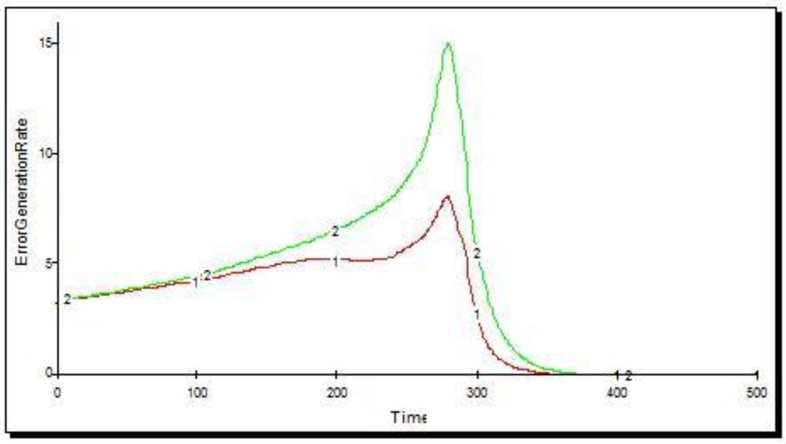


Figure 70. Errors generation rate,
Red line is before and green line is after cutting loopB1

From the first graph we see that by cutting loop B1, the balancing impact of this loop is removed. This is evident by the increase in the error generation rate, when there is no effect of learning from the design to coding phase. The effect of easier tasks due to the difference in errors type during design and coding is negated. Initially the behavior is similar. However, as the project progresses and moves toward easier design tasks, and coding tasks, the numerical difference is apparent. Without this loop however, R1 is dominant and its effect is increased causing the error generation to increase radically. So the behavior shows the same pattern, just at a much higher rate.

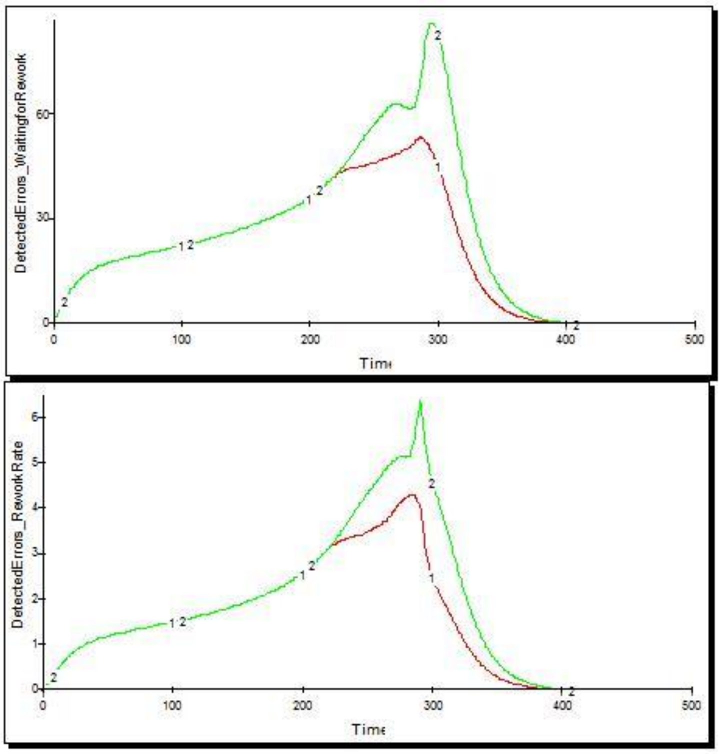


Figure 71. Detected errors waiting for rework and detected errors rework rate, Red line is before and green line is after cutting loopB1

From the two graphs that constitute the reference mode we see the same effect as described above. Since more errors are generated through the entire project, the stock of detected errors waiting for rework increases rapidly as the project progress and moves into the coding stage. In fact, the behavior that R1 is causing in the entire model is increased in numerical magnitude due to the cut of balancing effect of B1. The number of detected errors increases, and management re-allocates more manpower to rework these detected errors. As a result the rework-rate increases accordingly.

This graph does not provide an anomalous behavior like the previous cut. The behavior of the reference is similar here, but the levels are much higher. This is still violating the empirical data from the reference mode, so B1 is an important loop. This increases loop B1s validity.

7.2.6.3-Cutting loop R2

In the third loop knock-out, I remove the effect of loop R2. This loop manages the effort needed for detection activities through progress of the project and the percent of job actually worked. It also influences the error detection activities. The loop governs the relationship between allocation of manpower for software development, quality assurance manpower, manpower needed to rework the average error and the error detection activity. This is controlled through the progress of the project, difference in errors type and productivity due to learning.

This loop is cut by setting the "nominal manpower needed per average error" to a constant with the initial variable value. This negates the effect of progress of the project from design to coding phase, and productivity due to learning. The CLD is shown below:

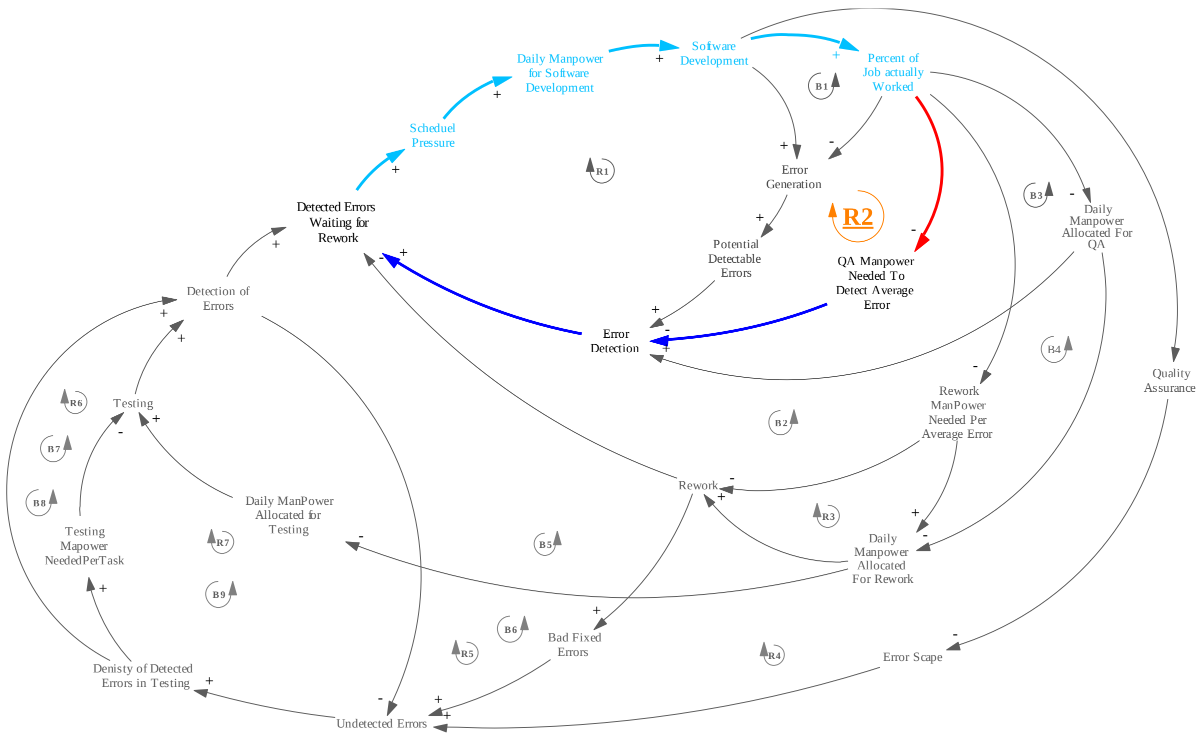

Daily Manpower for Software Development
Software Development
Percent of Job actually Worked
Scheduel Pressure
R1
B1
Error Generation
R2
Detected Errors Waiting for Rework
Potential Detectable Errors
QA Manpower Needed To Detect Average Error
B3
Daily Manpower Allocated For QA
Detection of Errors
Error Detection
Quality Assurance
B4
R6
Testing
Rework ManPower Needed Per Average Error
B2
B7
B8
Daily ManPower Allocated for Testing
Rework
R3
Testing Mapower NeededPerTask
R7
B5
Daily Manpower Allocated For Rework
B9
B6
Bad Fixed Errors
R5
R4
Error Scape
Denisty of Detected Errors in Testing
Undetected Errors

Figure 72. Quality assurance and rework and testing subsectors CLD, with cutting loop R2

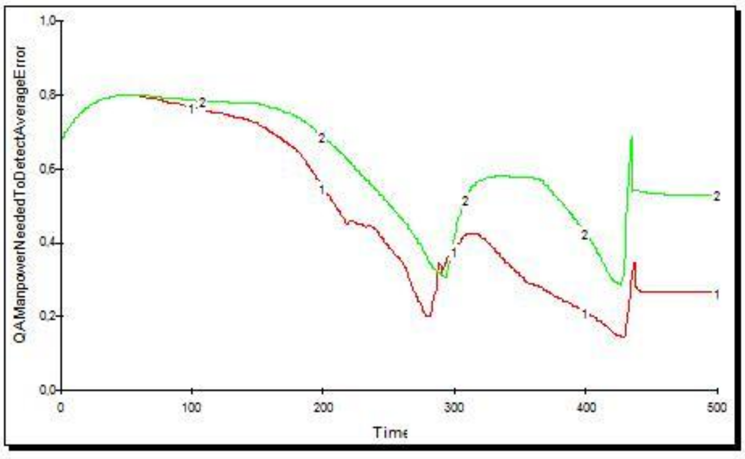


Figure 73. QA manpower need to detect an average error, Red line is before and green line is after cutting loopR2

From the first graph we see that the overall effect of the cut was not immediately significant. The behavior of the variable follows the initial behavior at a remarkably close pattern. The difference is mostly in the numerical range. By cutting loop R2 quality assurance manpower needed per average error follows the same pattern, just delayed and with a difference in numerical range. This negates the effect of learning from design to coding phase, and the effect of easier errors due to the difference in errors type during the design and coding phase. Initially the behavior is similar. However, as the project progresses, and moves toward easier design tasks and coding tasks, the numerical difference is apparent. Still this is an anomaly compared with the reference-mode, so loop R2 holds significance.

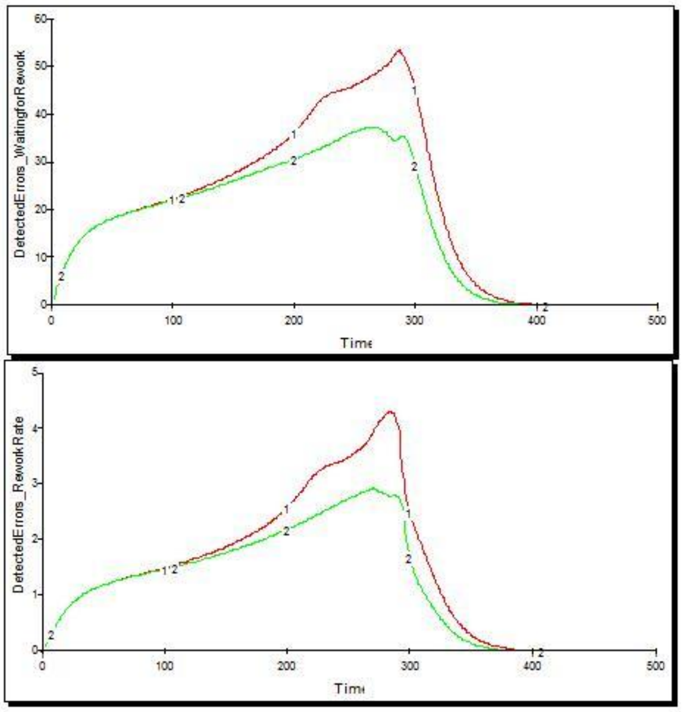


Figure 74. Detected errors waiting for rework and detected errors rework rate, Red line is before and green line is after cutting loopR2

In the reference mode we see that the effect of cutting loop R3 changes the behavior of the reference variables. The change reflects the strength of B1 without R2. The increase in detected errors waiting for rework, and the rework rate follow an increase-pattern. The magnitude however, is clearly different. The reason rests in the constant effect of percent of job done on the manpower needed to detect the average error. This constant leads to no dynamic effect by productivity due to learning and easier coding errors as the project progresses from design to coding.

As a result, as the project progresses, the effort needed per average error keeps a higher amount that causes a lower error-detection. Hence, the amount of errors that are detected and waiting for rework decrease and the effort put into reworking errors is subsequently lower. Management will allocate less manpower to the rework process as a result of the declined amount of errors detected waiting for rework. This is reflected in the graphs above. The final peak that is governed by R2 is also missing in the behavior observed in both graphs above. This strengthens the validity of loop R2.

7.2.6.4-Cutting loop B2, B6, R6

The fourth cut is the first cut where several loops get severed by one knockout. The loops that will be knocked out are B2, B6, and R6.

Loop B2 governs the rework activity through the rework effort needed per average error as project progresses. The loop controls this function through percent of job actually done, the rework-manpower needed per error, rework, detected errors waiting for rework and the allocation of manpower to software development.

Loop B6 governs the generation of bad fixed errors through the rework activity. This is done through the percent of job actually done, the rework manpower needed per error, the rework rate and bad fixed errors. The effect influences the density of detected errors in testing, the testing activity, the amount of detected errors waiting for rework and the manpower allocated for development.

Loop R6 governs the generation of bad fixed errors through the rework activity. The variable follows the same logic as B6, until the density of detected errors in testing. The effect is directly influencing the detection of errors waiting for rework and not indirectly like B6 through the testing effort. This is depicted in the CLD below. The loop is cut by setting “Rework Manpower Needed per Average Error” to a constant with the initial variable value.

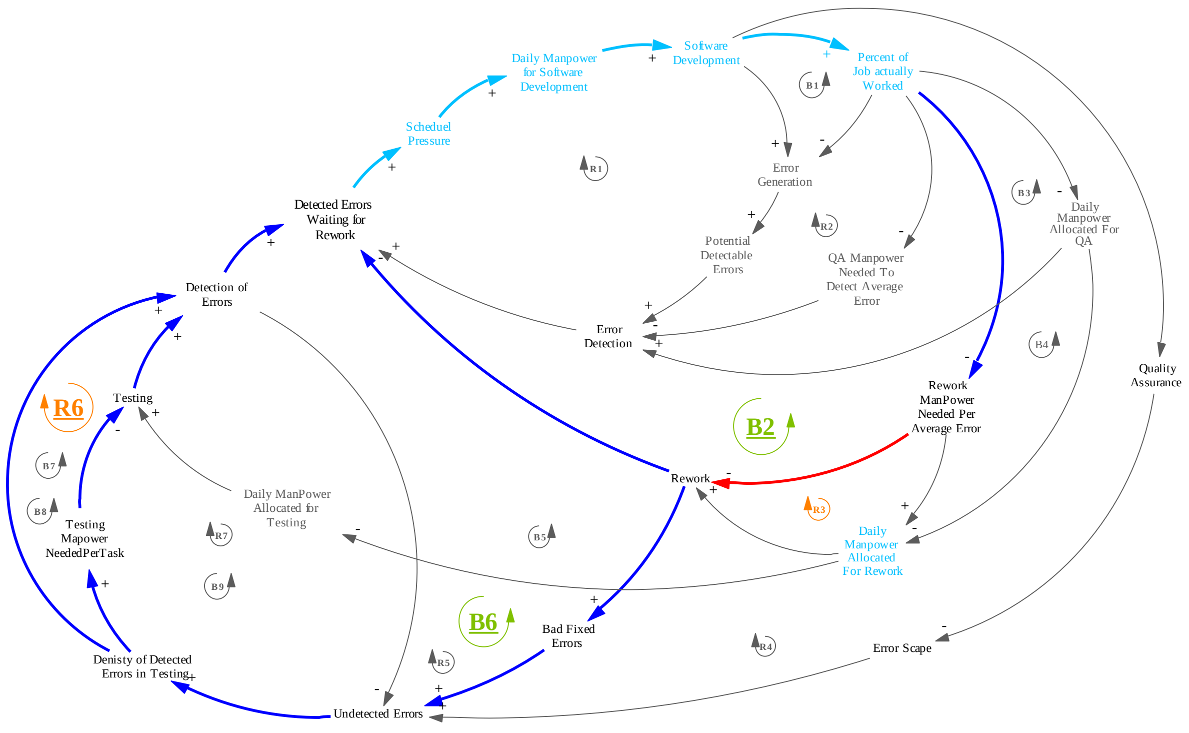

Software Development
Daily Manpower for Software Development
Percent of Job actually Worked
Scheduel Pressure
Error Generation
Detected Errors Waiting for Rework
Potential Detectable Errors
QA Manpower Needed To Detect Average Error
Daily Manpower Allocated For QA
Detection of Errors
Error Detection
Quality Assurance
Testing
Rework ManPower Needed Per Average Error
Rework
Daily ManPower Allocated for Testing
Testing Mapower NeededPerTask
Daily Manpower Allocated For Rework
Bad Fixed Errors
Error Scape
Denisty of Detected Errors in Testing
Undetected Errors
R1
R2
R3
R4
R5
R6
R7
B1
B2
B3
B4
B5
B6
B7
B8
B9

Figure 75. Quality assurance and rework and testing subsectors CLD, with cutting loops B2, B6, R6

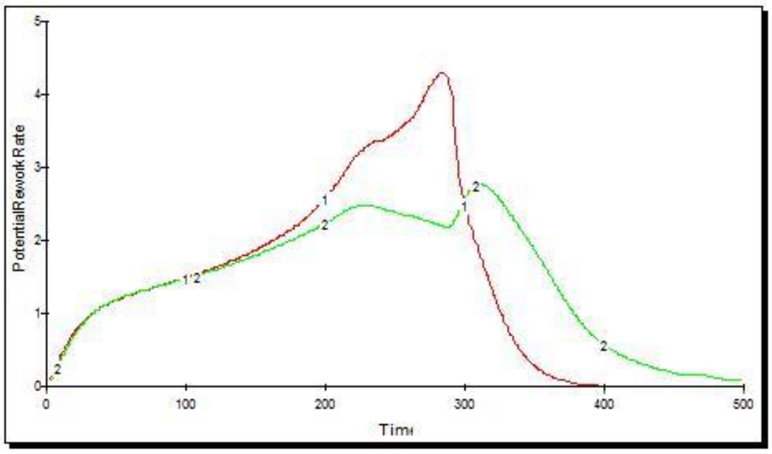


Figure 76. Potential rework rate,
Red line is before and green line is after cutting loops B2, B6, R6

The first graph shows that the loop-cut changes the behavior of the model near to the end of the design-phase. The potential rework-rate increases equally to the situation where the loops are not cut. As project progress, the constant rework manpower needed per average error on potential rework rate makes this rate smaller since learning and error-type is negated. The potential rework rate level does not increase in the same manner as before. The potential declines for a period of time in the project, until it peaks and slowly levels off to zero. The effort runs much longer. This is due to the lack of a dynamic effect by productivity due to learning and easier coding errors as the project progresses from design to coding. As the project progresses, the rework effort needed per average error is constant. No effect from progress is apparent as a result of the cut.

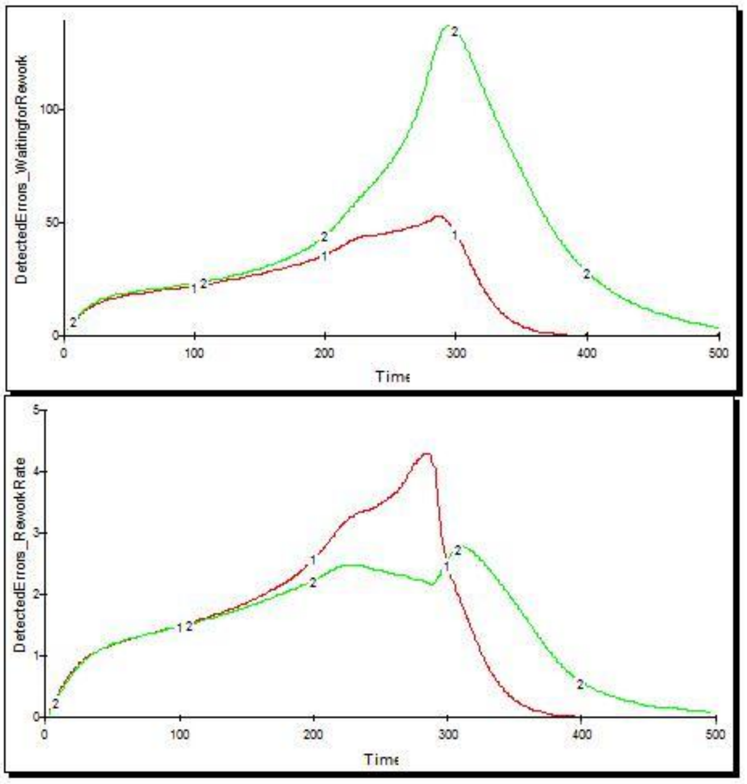


Figure 77. Detected errors waiting for rework and detected errors rework rate, Red line is before and green line is after cutting loops B2, B6, R6

By cutting these three loops, the models ability to reproduce the reference mode has been impaired as a result. This is visible in the two graphs above. The number of detected errors waiting for rework increases strongly, as the rework-rate is not able to respond to the errors waiting for rework. This is a combined effect of the exclusion of B2, B6 and R6.

In this way I have eliminated the feedback of bad fixes to the detected errors waiting for rework. Since the rework rate is no-longer governed by the percent of job actually worked and the rework-teams are not learning, and all errors are equally hard to rework. This provides the change in behavior observed above. This is due to the cut of B2, B6 and R6. The more significant loops of the three are B2 and B6, but the combined effect of all the loops are important for the system. The validity of all three loops has been strengthened by the observed changes in behavior.

7.2.6.5-Cutting loop R3, R5, B7

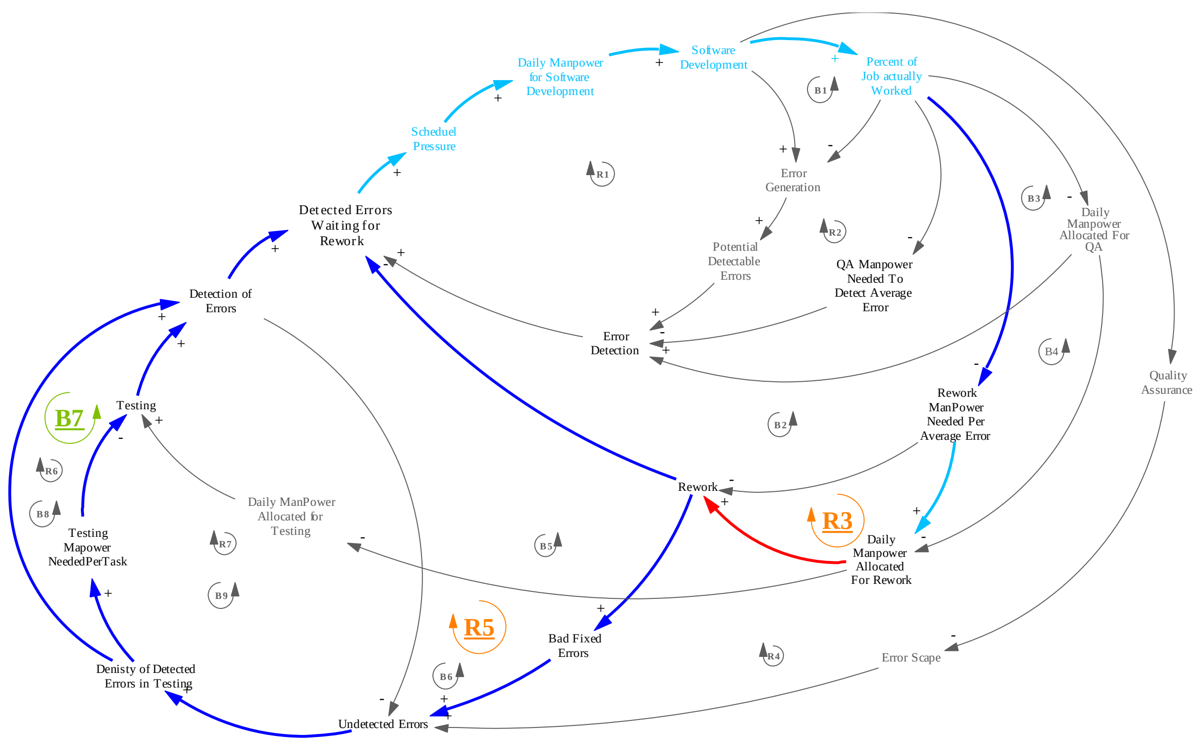
The fifth loop-cutting removes the effect of loops R3, R5, and B7. These loops are governing the rework activity through the daily manpower-allocation to the rework effort. By knocking this loop, we should see a pattern where the rework-effort is not increasing much during the production stage. In turn the detected errors waiting for rework should soar.

Loop R3 is governing the rework activity through daily manpower allocated for rework. This in turn is determined by the progress of the project and the percent of job actually worked through the rework manpower needed per average error.

Loop R5 governs the generation of bad fixed errors through the rework activity and daily manpower allocated for rework. The bad fixes influence the detected errors are waiting for rework through the density of detected errors in testing and the testing activity.

Loop B7 is following the same logic as R5, but is concerned with the direct effect of error density on the detection of errors in testing and likewise on detected errors waiting for rework. The R5 loop holds this effect indirectly through the testing effort. This is shown in the CLD below.

The loops are cut by replacing the “Daily Manpower Allocated for Rework” variable with a constant. The constant holds the initial value of the variable.



Daily Manpower for Software Development
Software Development
Percent of Job actually Worked
Scheduel Pressure
Detected Errors Waiting for Rework
Error Generation
Potential Detectable Errors
QA Manpower Needed To Detect Average Error
Daily Manpower Allocated For QA
Detection of Errors
Error Detection
Quality Assurance
Testing
Rework ManPower Needed Per Average Error
Rework
Daily ManPower Allocated for Testing
Daily Manpower Allocated For Rework
Testing Mapower NeededPerTask
Bad Fixed Errors
Error Scape
Denisty of Detected Errors in Testing
Undetected Errors
B1
B2
B3
B4
B5
B6
B7
B8
B9
R1
R2
R3
R4
R5
R6
R7

Figure 78. Quality assurance and rework and testing subsectors CLD, Red line is before and green line is after cutting loops R3, R5, B7

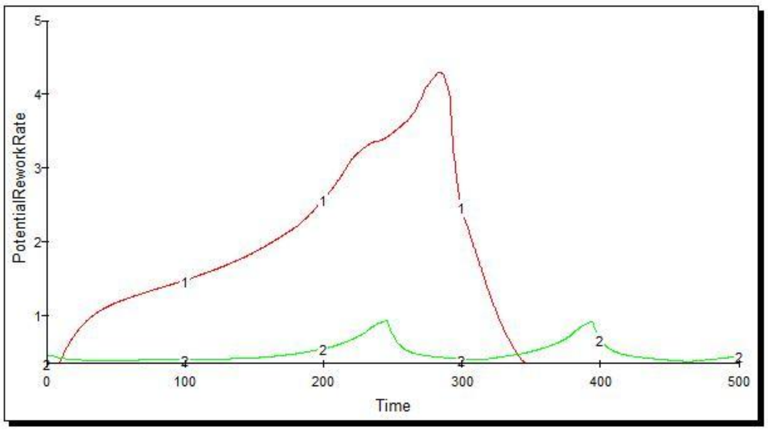


Figure 79. Potential rework rate, Red line is before and green line is after cutting loops R3, R5, B7

From the first graph we see the impact of cutting these three loops straight away. The dynamic allocation of manpower for the rework-effort is turned off, and the potential for rework activity holds a very low rate. This is a large impact, and the behavior is far from reproducing the base-behavior by having very low rate and it's delayed. We see that there are some increases, visible as peaks in the rework effort, which are equal to the behavior before cutting the loops. This behavior is governed by loop B2. This loop influences the rework effort through the manpower needed per average error. Hence, for some periods of time the effort is increasing, but as the graph shows the reinforcing loops of the system are generally more dominant. This is anomalous behavior as described by Sterman.

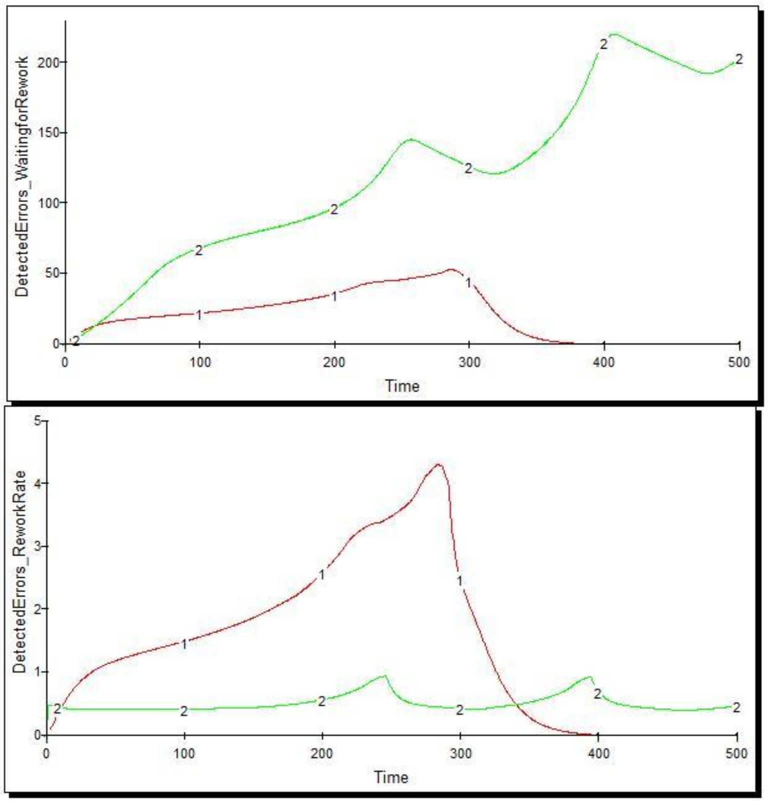


Figure 80. Detected errors waiting for rework and detected errors rework rate, Red line is before and green line is after cutting loops R3, R5, B7

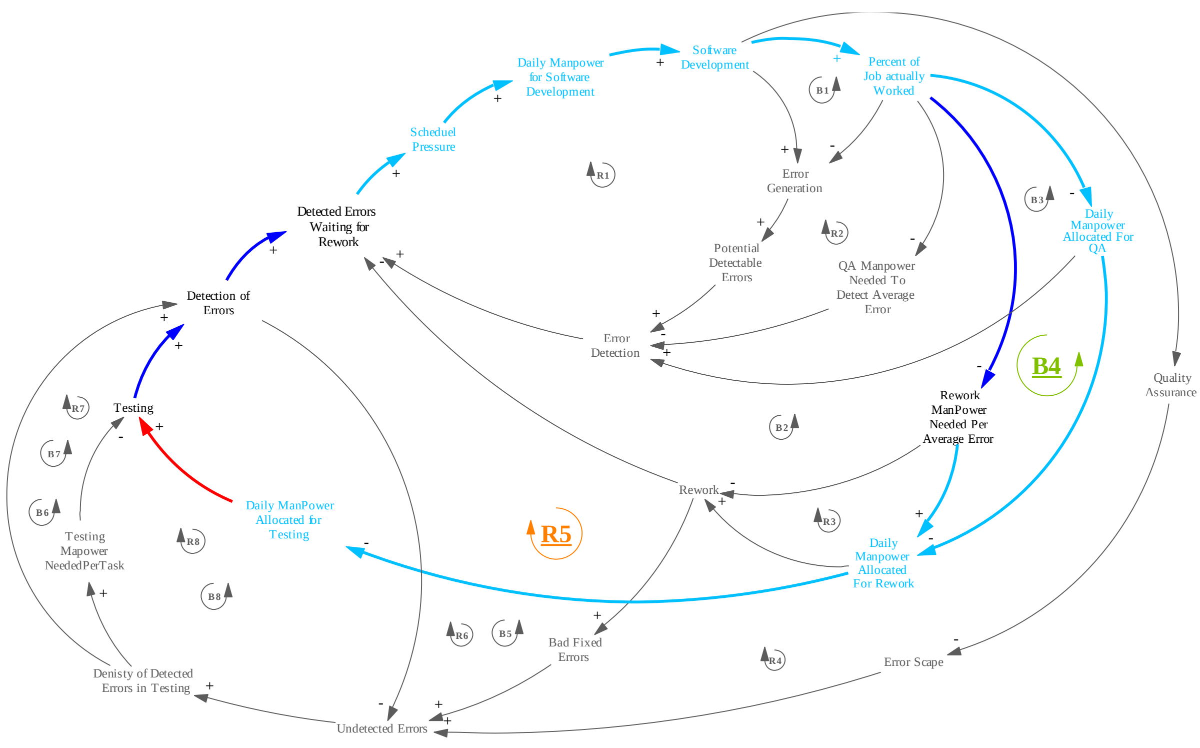
The significant change in behavior for the previous variable provides as expected changes for the reference mode as well. The number of detected errors waiting to be reworked increases strongly, as the rework effort is not increasing. The behavior of the graph does not follow real-world logic anymore, as the stock of detected errors fall at a certain point in time as well. This decrease occurs even though the rework effort is not increased in any significant manner. Based on these changes it is very clear that the cut loops are important, and that excluding them renders us unable to reproduce the reference mode. By the changes in the graph is apparent that loop R3 and R5 are the most significant, but that all the loops play a vital role

7.2.6.6-Cutting loops B4, R5

The sixth knockout procedure removes the effects of loop B4 and R5. These two loops are governing the testing effort through the daily manpower allocated for testing. This effect will now be removed. By cutting these loops I expect the testing-effort to halt completely, and keep to a zero-level through the project cycle. By cutting this loop I practically induce the same effect as having one testing phase with no feedback to the quality assurance and rework effort.

Loop R5 governs the testing effort as project progresses by the percent of job actually through the rework-manpower needed per error. This influences the daily manpower allocated for rework, the daily manpower for testing and the testing effort. As we explained before in chapter 6 production manpower will be allocated between development, rework and testing. Therefore increase in allocation of manpower for one effort leads to decrease of manpower allocated to another. Testing influences the detection of errors in testing and the stock of detect errors waiting for rework.

Loop B4 governs the testing effort as project progresses by the percent of job actually through daily manpower allocated for quality assurance and daily manpower allocated for testing. The loop follows the same logic as loop R5, but is balancing the system as manpower allocated for rework increases, as the manpower allocated for quality assurance decrease by the progress of the project. This leads to a decrease in daily manpower allocated for the testing effort. The relationship is shown in the CLD below. I cut the loop as before, by replacing the variable “Daily Manpower for Testing” with a constant holding the initial value that is zero.



Daily Manpower for Software Development
Software Development
Percent of Job actually Worked
Scheduel Pressure
Detected Errors Waiting for Rework
Error Generation
Potential Detectable Errors
QA Manpower Needed To Detect Average Error
Daily Manpower Allocated For QA
Detection of Errors
Error Detection
Quality Assurance
B4
Testing
Rework ManPower Needed Per Average Error
Daily ManPower Allocated for Testing
R5
Rework
Testing Mapower NeededPerTask
Daily Manpower Allocated For Rework
Bad Fixed Errors
Error Scape
Denisty of Detected Errors in Testing
Undetected Errors
R1
R2
R3
R4
R6
R7
R8
B1
B2
B3
B5
B6
B7
B8

Figure 81. Quality assurance and rework and testing subsectors CLD, with cutting loops B4, R5

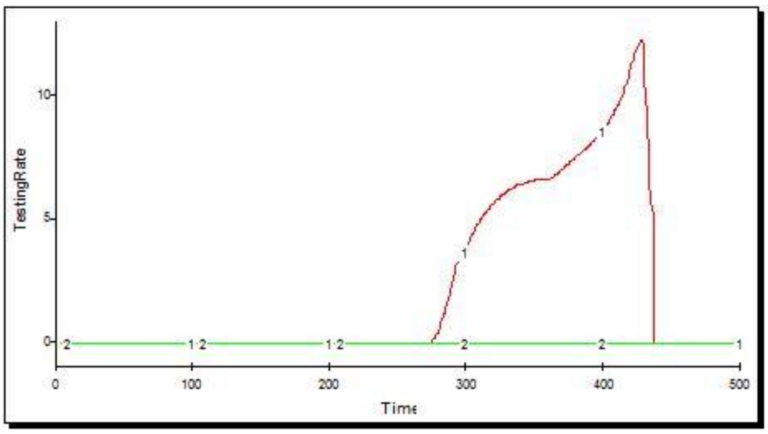


Figure 82. Testing rate,

Red line is before and green line is after cutting the loops B4, R5

The first graph shows the expected result of cutting the loops. The testing-rate is zero through the entire software project. The project continues to run as no tasks are being tested, and there is not a single bump in the curve along the way. The behavior of the variable has changed completely as a result of cutting the two loops. From the graph we see in graph above that the reinforcing and balancing loop creates increase and level-off in the testing effort during the project. By cutting these loops the effect is completely gone.

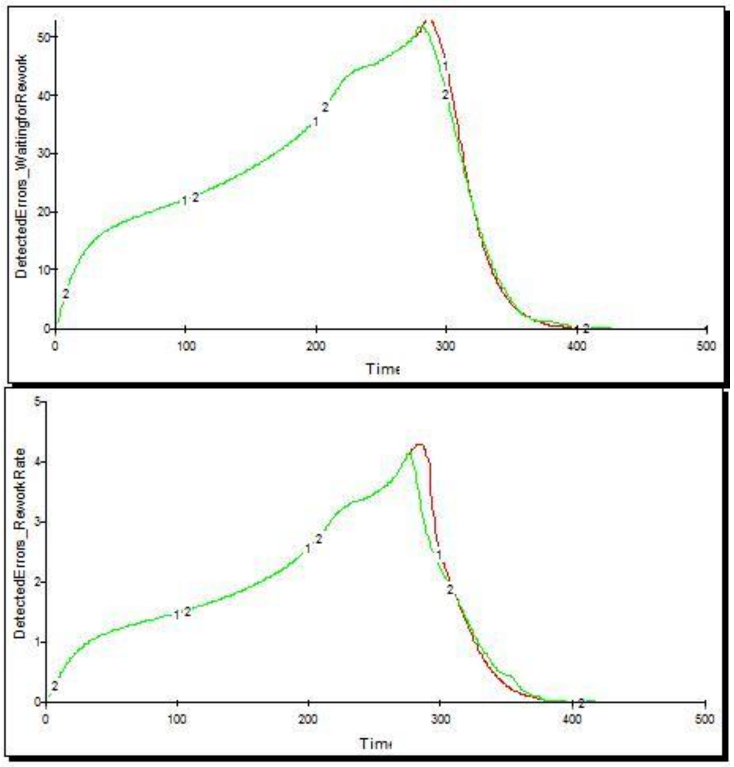


Figure 83. Detected errors waiting for rework and detected errors rework rate, Red line is before and green line is after cutting the loops B4, R5

For the reference-mode it is interesting to see that the effect in both of the graphs above appears near to the end of the project. This is at the end of the coding phase when nearly all of the tasks are developed and production manpower will be re-allocated for the testing effort. Detected errors in testing are fed back to rework to get corrected. By cutting the loops, there are no resource allocated for the testing effort and no active systems testing. Likewise, no detected errors in testing are added to the stock of detected errors waiting for rework to get corrected. Cutting these loops proves that applying the feedback from the testing effort to production is an important enhancement for realism in the project as described by the literature on the field. Therefore these two feedback loops are important loops, and should be included in the model.

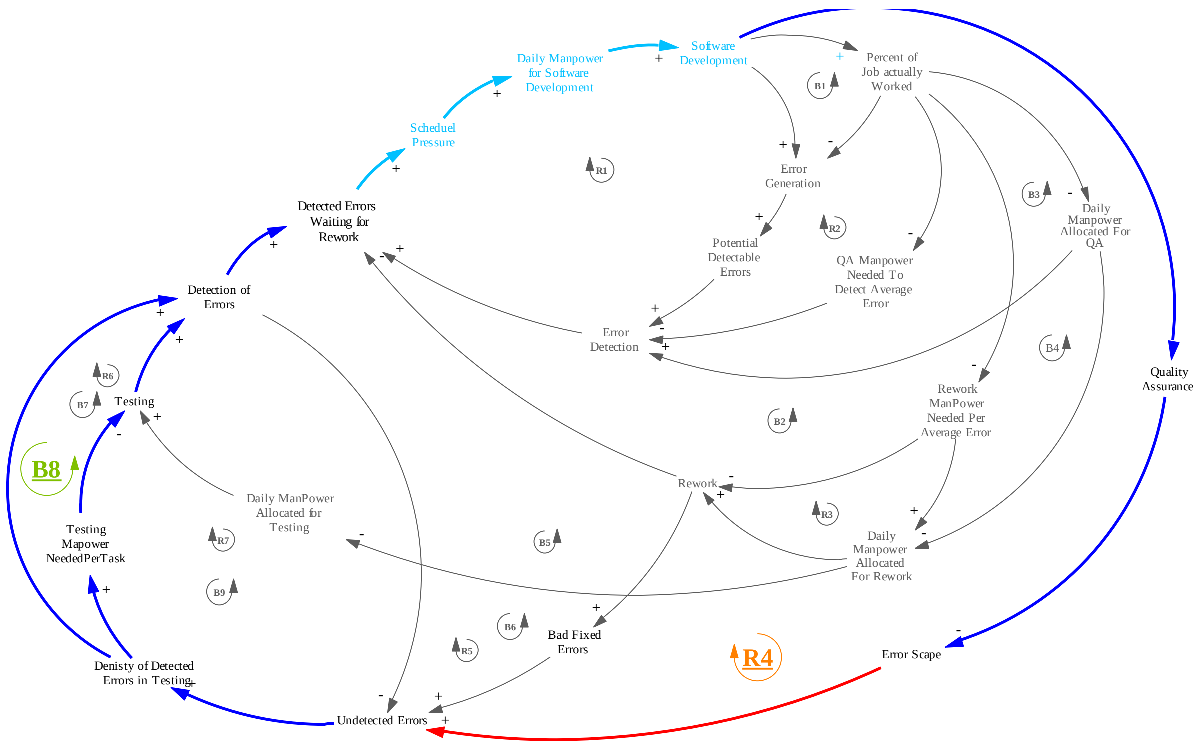
7.2.6.7-Cutting loop R4, B8

The seventh loop-cutting removes the effect of loop R4 and B8. These loops concern the escaped errors from quality assurance on the testing effort. This is done through the development of software and the quality assurance effort. By cutting these loops we expect to see a significant reduction in detected errors in testing. The effort is now finding every single error, and should have an impact on the overall detected errors in testing. The only errors occurring for the testing effort without escaped errors stem from bad fixes which holds an average of 7.5% approximately according to the literatures.

Loop R4 controls the testing effort through the undetected errors that occur due to escaped errors. These errors influence to the total density of detected errors in testing, and the testing-effort through testing manpower needed per task. This in turn influences the detected errors waiting for rework through error detection in testing, and the software development rate through daily manpower allocated for development.

Loop B8 follows the same line of variables as loop R4, but deviates as the density of detected errors in testing directly influence the error detection in testing rather than through the manpower needed per task and testing rate.

This relationship is depicted in the CLD below. The loop is once again cut with replacing the original variable with a constant holding the initial variable value. In this case it is the "Error Escape Rate" that is held constant at the value of zero.



Software Development
Daily Manpower for Software Development
Scheduel Pressure
Detected Errors Waiting for Rework
Detection of Errors
Testing
Testing Mapower NeededPerTask
Denisty of Detected Errors in Testing
Undetected Errors
Percent of Job actually Worked
Error Generation
Potential Detectable Errors
QA Manpower Needed To Detect Average Error
Error Detection
Daily Manpower Allocated For QA
Quality Assurance
Rework ManPower Needed Per Average Error
Rework
Daily Manpower Allocated For Rework
Daily ManPower Allocated for Testing
Bad Fixed Errors
Error Scape
R1
R2
R3
R4
R5
R6
R7
B1
B2
B3
B4
B5
B6
B7
B8
B9

Figure 84. Quality assurance and rework and testing subsectors CLD, with cutting loops R4, B8

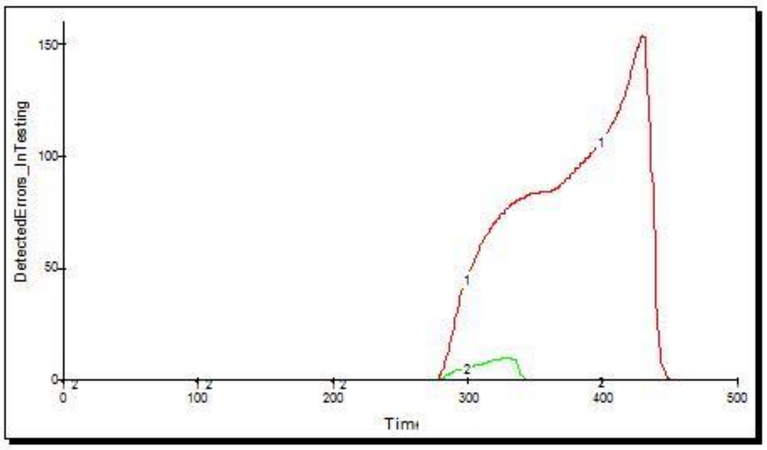


Figure 85. Detected errors in testing,
Red line is before and green line is after cutting the loops R4, B8

From the first graph we see an instant change in behavior for the variable. By removing the effect of escaped errors, the amount of errors detected in testing is significant. The only errors that are detected in this effort at the moment stems from bad fixes, and they only constitute

7% on average in a software project. This is as predicted in the introduction to the seventh loop-cut. This changes the behavior of the initial graph, and we also observe the combine effect of both loop R4 and B8.

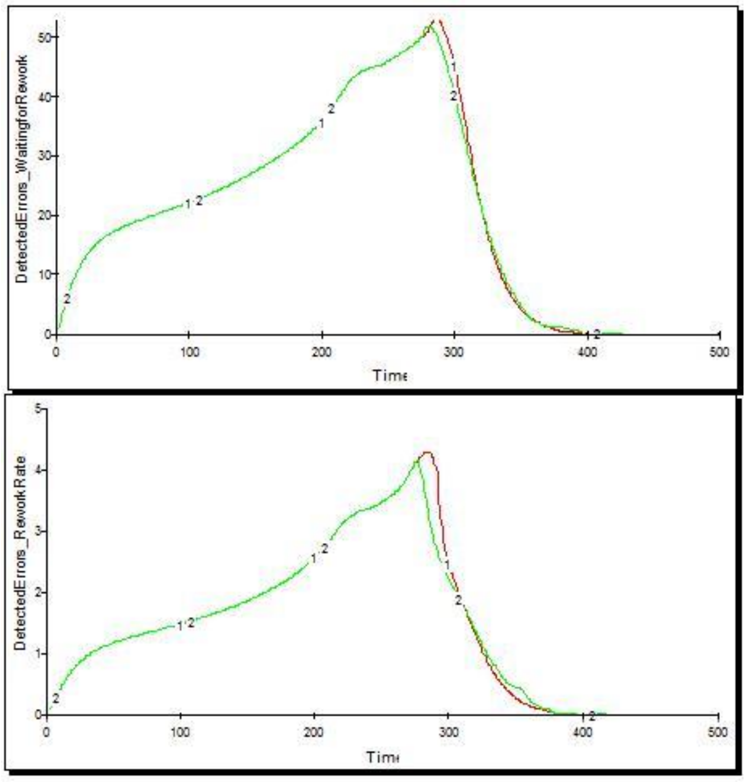


Figure 86. Detected errors waiting for rework and detected errors rework rate, Red line is before and green line is after cutting the loops R4, B8

For the reference-mode it is interesting to see that the effect in both of the variables is present close to the end of the project. The effect occurs at the end of the coding phase when nearly all of the tasks are developed, and production manpower is moved to testing. Detected errors in testing are fed back to the rework effort for correction. By cutting these loops there are no escaped errors to be detected in testing. Consequently fewer detected errors in testing are added to the stock of detected errors waiting for rework, since the testing effort brings some additional errors that stem from bad-fixes. The behavior of both graphs follows the same trajectory minus the effect of escaped errors that are visible for the first run. The result is a sharper decline in run 2.

From both graphs it’s clear that the B8 loop is the dominant loop, but the R4 loop holds some effect. By the influence of these loops on the reference mode the importance of the testing effort for real-world systems have been established. This loop-knockout proves that applying feedback from the testing effort to production is an important enhancement for realism in the project as described by the literature. Therefore these two feedback loops are important loops, and should be included in the model.

7.2.6.8 Cutting Loop R5, R6, B6, B7

The last and final cutting-procedure investigates the opposite relationship from cut seven. In this case the bad-fixes are eliminated as an effect. I predict that this should yield a situation where the detected errors in testing follow the same pattern as the original run, minus the effect of bad fixes. This will not have a huge effect, as the bad fix rate is normally at 7% on average. Just as observed in cut seven, the changes should be less. Behavior-pattern should be equal, but numerically a bit different.

Loop B6 influences the undetected errors for testing through the bad fixed errors. This is provided through the percent of job actually worked, the rework manpower needed per average error and its direct influence on the rework effort.

Loop R5 follows the same logic and pattern as B6. It governs the bad fixes through the rework effort, but unlike B6 through the daily manpower allocated for the rework process. The rework effort influences the bad fixes generation, and subsequently the undetected errors. Both loops B6 and R5 influence the detected errors waiting for rework through the density of detected errors in testing, testing manpower needed per task and testing effort.

R6 follows the same logic and pattern as B6. The difference in loop R6 is the fact that density of detected errors in testing directly influences the detection of errors in testing, unlike B6 that runs through the testing effort.

B7 follows the same pattern as R5. The bad fixes are governed by the rework manpower needed per average error through the daily manpower allocated to the rework effort. Then as bad fixes leads to undetected errors, the density of detected errors in testing is increased. The error density unlike for loop R5 directly influences the detected errors in testing. Both loops B7 and R6 influence the detected errors waiting for rework through the density of detected errors in testing and errors detection in testing.

This relationship is depicted in the CLD below. I cut loops R5, R6, B6, and B7 through cutting the effect of the bad fixes. Just as in the other knock-outs, the loop is cut by trading the variable for a constant with the original value that is zero.

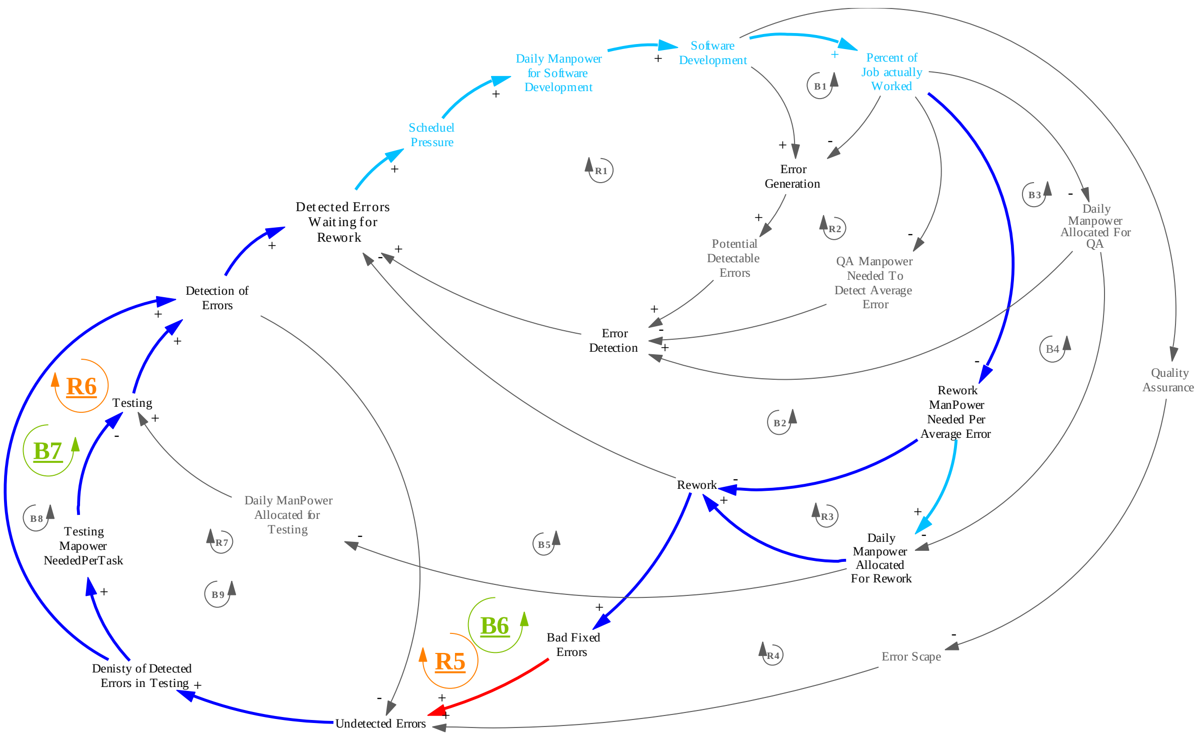

Software Development
Percent of Job actually Worked
Daily Manpower for Software Development
Scheduel Pressure
Detected Errors Waiting for Rework
Error Generation
Potential Detectable Errors
QA Manpower Needed To Detect Average Error
Daily Manpower Allocated For QA
Quality Assurance
Detection of Errors
Error Detection
Rework ManPower Needed Per Average Error
Testing
Daily ManPower Allocated for Testing
Rework
Daily Manpower Allocated For Rework
Testing Mapower NeededPerTask
Bad Fixed Errors
Error Scape
Denisty of Detected Errors in Testing
Undetected Errors
R1
R2
R3
R4
R5
R6
R7
B1
B2
B3
B4
B5
B6
B7
B8
B9

Figure 87. Quality assurance and rework and testing subsectors CLD, with cutting loops R5, R6, B6, B7

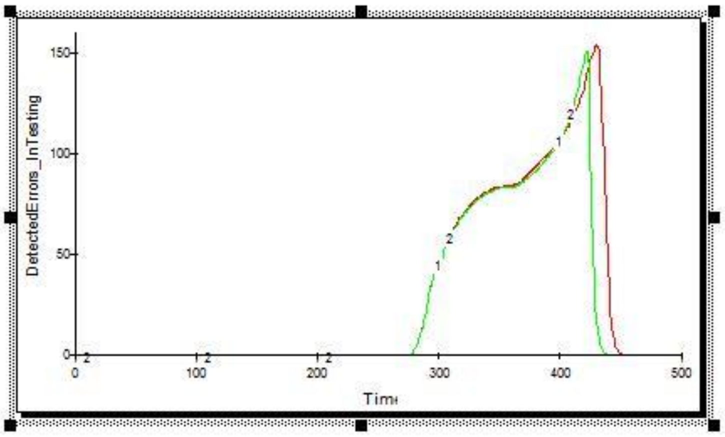


Figure 88. Detected errors in testing, Red line is before and green line is after cutting the loops R5, R6, B6, B7

From the first graph we see a negligible change in behavior for the variable. By removing the effect of bad fixed errors, the amount of errors detected in testing is miniscule. The errors that are detected in this effort at the moment, stems from escaped errors that are considerable. This is as predicted in the introduction to the eighth loop-cut. This changes the behavior of the initial graph, and we also observe the combine effect of both loop R6 and B7.

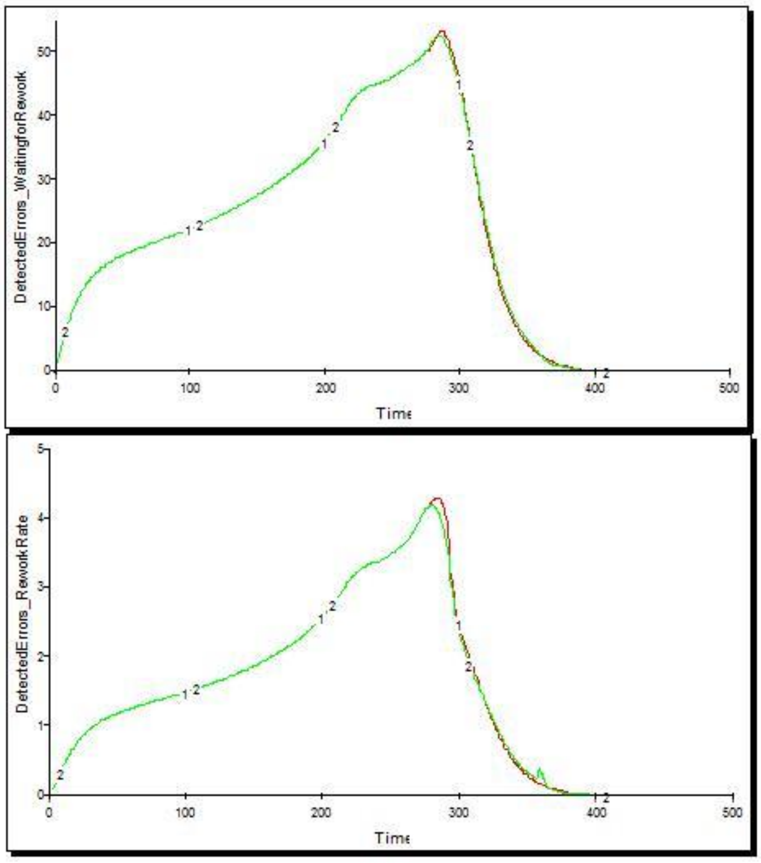


Figure 89. Detected errors waiting for rework and detected errors rework rate, Red line is before and green line is after cutting the loops R5, R6, B6, B7

For the reference-mode it is interesting to see that once again the effect shown in both of the graphs above, appears near to the end of the project. It manifests itself at the end of the coding phase when nearly all of the production tasks have been completed. As more manpower is moved to testing, more errors are detected in testing and fed back to rework. By cutting these loops there are no bad fixed errors to be detected in testing and as a result, less detected errors in testing are transferred to the stock of detected errors waiting for rework to get rework since the testing effort brings additional errors that stem from escaped errors. The behavior of both graphs follows the same trajectory minus the negligible effect of bad fixes that are visible for the first run.

From both graphs it’s clear that the combine effect of R6 and B7 loops is dominant, but R5 and B6 loops holds some effect. Even though the effect of these loops is minor, but through the influence of these loops on the reference mode the effect of the testing effort for real-world systems have been established. This establishes yet again the importance of feedback between testing and the rework effort. The cut of these loops show their importance to the overall behavior of the model, and they should be included to provide the needed realism to real-world projects.

Chapter 8: Policy Analysis

8.0 Introduction

In this chapter I will present different policies that management can pursue in order to achieve a cost-beneficial end-product and obtain a desired level of software quality. In this paper I have established the model-enhancements, and validated the model as a whole. Since the validation process has been completed, the model is soundly based on a theoretical and empirical ground. The model is able to reproduce the reference mode, and with this foundation I can start to formulate managerial policies.

The strength of a system dynamics model is the ability to reproduce graphically or numerically the characteristics of a problem. In my case, the dynamics behind defect removal and software development. When a model is able to reproduce the observed problem, and it has been thoroughly validated, we are able to conduct model-experiments in order to find policies that can solve the problem or improve the overall behavior of a system. In my case the aim is to provide management with policies that provides better defect-removal at a cost-beneficial rate. It is important to note that I will not decide on which policies are more suitable for software projects. Each software project is unique in regards to content, width and scope. The production strategy is a managerial decision made between software development management and the clients. My model is a tool where different policies can be constructed, that will be an aid for management and clients in the pre-production planning stage. Through the paper I have identified that software quality hinges on three main efforts, namely the quality assurance effort, the rework effort, and the testing effort.

The aim of my paper is to formulate different policies that can aid management to incorporate these efforts in order to improve the end-product. From the software crisis it is apparent that this is a challenge, and it rests on the managerial side of software project. Through this chapter I will first present two policy strategies for each effort and compare it with the original base-run. When these policies have been identified, I will combine them in a comprehensive analysis of synthesized policies. It is of vital importance that the dynamic relationships between each effort are incorporated in my suggested policies. As identified by Abdel-Hamid, the software development process is complex. If one creates isolated policies, they will have little value for management. This is due to the fact that all branches of production are interconnected, and that a change in policy for one process will have an effect on all the others as well. Therefore, this chapter provides several combinations of policies for management that takes the dynamic relationships into consideration.

8.1 Policy domains and analysis

The next part of this chapter presents the policy domains and the basis for my analysis of the policies I have formulated in order to provide better software quality. As I mentioned in my introduction, my paper have identified three key efforts for this endeavor. They are: quality assurance, rework, and testing.

Quality assurance is the pre-test defect removal tool that continuously monitors the finished tasks from software production. Quality assurance staff performs low-level tests, and hold a dialogue with the production team as the software progresses from design to coding. The quality assurance effort is on the micro-level and concerns the code-structure. This effort provides valuable information on tasks and progress. With the capture-recapture effort the quality assurance team provides a second cooperation-function with testing.

The rework effort is teams of staff that correct errors found by quality assurance and testing. The rework team is designated workers from the software production staff, and they hold a vital function as errors and weaknesses are removed from the product before completion.

The testing effort consists of a series of tests on finished applications as the project progresses and tasks are completed. These tests are at a higher level than the quality assurance effort, and provide the meso-level of the software production effort. In testing multiple tasks set together as a functional application is tested, and analyzed. (Jones and Bonsignour 2012) This effort in combination with quality assurance is an effective defect-removal tool. In addition, a client-review system for this effort removes additional deviations, and ensures software quality.

These important efforts are all driven by the allocation of manpower. The true engine behind the performance for software production, defect-removal, and error correction is managerial allocation of manpower in a project. For every software project, there is a limited amount of manpower, technical supplies and equipment that must be allocated between different efforts.

In our case it is the manpower-allocation that is paramount, and this is reflected in the policy design for my policy suggestions presented in this chapter.

In order for software to be produced, for errors to be identified and corrected, and for testing of applications manpower needs to be allocated. It is very important for management to understand and handle these three efforts in tandem with the software development effort. From the software crisis, it is evident that mismanaging these efforts can lead to serious problems. A software project can be severely delayed, or even stopped as a result. In other cases the software has been completed behind schedule, and still found to contain grave errors that render it useless. The software industry is hampered with these cases, and it's the leading industry in negative quality issues.

As a result, my policies are designed in the following manner: The first policy is the normal base-run policy provided by Abdel-Hamid. Then I introduce two new policy-strategies that allocate manpower at two different levels. I then compare the results from the two policy-strategies to the base run. By utilizing this approach, I will be able to provide a range of policies for all efforts that is applicable for software projects. The policy that management and client in the end chose to incorporate is based on the project's frame, and client's wishes.

The three policy factors that govern the allocation of manpower are the variables that control manpower allocation for the three efforts. They are: (1) Desired Rework Delay which is a constant, (2) Planned Fraction Of Manpower For Quality Assurance which is a graph function, and (3) Fraction Of Effort For System Testing which is a graph-function. These policy variables will all be presented in greater detail during the policy-analysis.

The first policy variable is called "Desired Rework Delay". This policy variable controls the amount of days desired to spend on the rework-effort when an error has been detected. When management plans for error correction, they introduce a desired elapse of time from an error is identified to its reworked and corrected. The amount of manpower allocated to rework hinges therefor on this desired rework delay.

The second policy variable is called Planned Fraction Of Manpower for Quality Assurance. This policy variable controls the fraction of total daily manpower after training that is designated to the quality assurance effort. It is important to remember that quality assurance staff is delegated away from software production. The quality assurance staff remains tied to the quality assurance effort until the final system testing after all tasks has been produced. Hence, changes in the fraction of manpower allocated to quality assurance changes the manpower allocated to the effort.

The third and final policy variable is the Fraction of Effort for System Testing. This policy variable is a vital control for the testing effort in terms of both manpower and design. In my enhanced model, I introduced a continuous system of testing. The allocation of manpower to testing does not only govern the total amount of manpower dedicated to the effort, but also controls the ability to hold multiple testing stages. If manpower is allocated only in the end of a project, the number of testing stages is very few. If manpower is allocated early in a project, it is possible to have multiple testing-stages through the software production cycle. The amount of allocation and the time it's allocated dictate the design and efficiency of testing. This is a very important relationship.

Through these policy variables and policy relationships, I will be able to construct two policy-strategies for management to pursue for each effort. The goal is to provide policies that will enable management to obtain the highest degree of software quality, within a cost-beneficial framework. These strategies will vary from production oriented approaches, to more quality oriented approaches. There is a constant need for balance between these two aspects for software project management. In order to determine the effect of my policies, my policy analysis is linked directly to the reference mode. The reference mode depicted below shows the important stock and rate for detected errors waiting for rework, and the detected errors rework rate. Through chapter 5 and 6 I explained how these two graphs depict the challenge facing management in regards the quality of the software production process and the quality of the end-product. For my policy analysis the reference mode is the yard-stick.

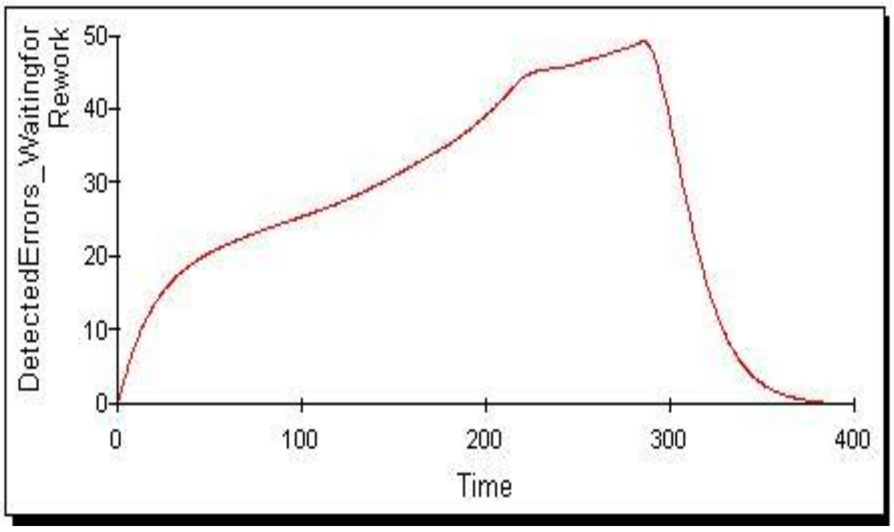


Figure 90. Detected errors waiting for rework

The amounts of errors are detected and waiting to be reworked varies through the phases of software production. As described in chapter 5, these variations are a result of different stages and processes during development, manpower-allocation, productivity, motivational factors, staff-mix and schedule pressure.

When a software project is established, management allocates their resources to the three major efforts undertaken in the early production phase. The first allocation is made to the quality assurance effort. The second allocation management performs, is the allocation to software development, and the rework effort. The third and final allocation is made to the testing effort. As mentioned above, there are in total six factors that contribute to the depicted behavior in the graph above. These factors have been presented in detail in chapter five, but I will briefly re-introduce them here. These factors are important for my constructed policies as well, and the results from the analysis reflect the relationships between production and factors.

When the software project has started, the production team starts to develop software. The tasks that are produced and completed are sent to the quality assurance effort, where errors are detected. While tasks are developed, errors are generated as well. Manpower allocated for quality assurance effort detects errors; these detected errors are sent to the rework effort in order to have them corrected. This is depicted in the graph above as the total number of detected errors waiting for rework increases sharply at the beginning of the project. Due to the nature of design errors, the initial effort to rework an average error is high. This changes during the course of the software project as it moves from design to coding. The graph-line levels off after a while, as errors are being reworked. This reduction in detected errors waiting is linked to the above-mentioned factors.

The first factor for the development is the learning factor and experience. In the beginning of a project, the staff is completely new for the new project. Some of them might have experience for other projects, but in a new project they also start from scratch as no project are completely similar (Abdel-Hamid, 1991: p.44). When the project progresses, more tasks are accomplished and the product is taking shape. The staff gains knowledge of the inner workings of their software as the different design aspects are established. Therefore, error generation during production become less and when quality control identifies errors, the rework teams have more knowledge about the design and how the tasks should operate. Hence, correcting such mistakes takes less effort due to increased rework experience which leads to a higher rework rate.

The second factor is development and progress that is presented through percent of job actually worked. When software is developing, the project progresses and moves from design phase to coding of development. The tasks that need to be completed are now easier to accomplish. In addition the increase in productivity due to learning along the project progress provides positive motivational factors. The feeling of accomplishment leads to better craftsmanship, and fewer errors.

The third factor is manpower-allocation for rework effort. When the stock of detected errors waiting for rework increases at the beginning of the project, it signals to management that allocation for rework is needed. Since design errors are harder to rework, the effort needed per error is higher. Management would then re-allocate staff to the rework effort, subsequently leading to more errors being reworked and a higher rework rate.

The fourth factor for the lower level of increase in detected errors waiting for rework in the beginning of the project lies in manpower allocation to the quality assurance effort. When more errors are detected and waiting for rework, parts of the development staff is utilized for rework. This will lead to a loss of production, and management perceives a progress-delay. If this delay is severe enough and under a schedule pressure situation, one solution would be to focus on production and suspend quality assurance. This is a practice found in empirical examples provided by Abdel-Hamid, where the quality assurance effort could be suspended for weeks or even stopped indefinitely. Errors are impossible to prevent, and the size of the quality assurance team provides the potential for detecting errors. Reducing the quality assurance effort equals a reduction in detected errors waiting for rework.

The fifth factor for the lower level in increase stems from motivational factors. Motivational losses and turnovers influence both the quality assurance staff and the development staff as well. Motivational losses reduce the potential software development rate and the potential error detection rate. Turn-overs translate into loss of experienced personnel that is replaced by inexperienced personnel. This result in fewer tasks completed and fewer errors detected. This would lead to a decrease in the necessary rework-rate, causing management to adjust the effort. This causes the effort to increase at a slower rate.

In the graph above there is also a peak after a time-period of smooth increase in the errors waiting for rework. There are three factors that explain this development. The first factor is the increase in production factor. As the software project is proceeds from design to coding, the overall productivity increases. The staff is able to complete more tasks at a higher rate, and the total number of tasks controlled by the quality assurance team increases. When more tasks are being completed, the chances of making errors increase as well.

Staff that has been working on the same project for a while tends to lose motivation over the course of production. Losses to motivation stems from tiredness, schedule pressures and other psychological factors. These motivational factors lead to more errors being made towards the end of the project, and constitute the second factor for the increase and peak.

The third factor is the turnover factor. As a project moves from design to coding, parts of the original workforce will quit or be fired. The software industry holds one of the highest turn-over averages of all industries. When people decide to quit their job or are fired, one experienced worker is lost and an inexperienced worker must take his place. This leads to both a drop in productivity, and an increase in error-proneness of the staff.

The fourth and final factor that explains this increase in detected errors waiting for rework towards the end of the project lies in quality assurance effort itself. If the quality assurance effort has been postponed for a time, the increase in the quality assurance effort will lead to an increase in detected errors. Secondly, as the software project moves from the design phase to the coding phase errors are easier to detect. When errors are easier to detect the potential for error-detection increases and more errors are detected and waiting for rework as a result.

The final pattern of the rework-rate is a decent in the effort towards zero. In this model, the testing effort is initiated after 80% of all tasks have been completed. Since the project is moving towards the final systems-test, manpower is re-allocated for the testing effort. Management re-route manpower from the development staff, quality assurance staff and the rework staff. However, since there are still errors waiting for correction the transfer of manpower from rework is done incrementally. This constitutes the slow decrease in the rework-effort as a steadily smaller team of rework-personnel completes the remaining error correction tasks.

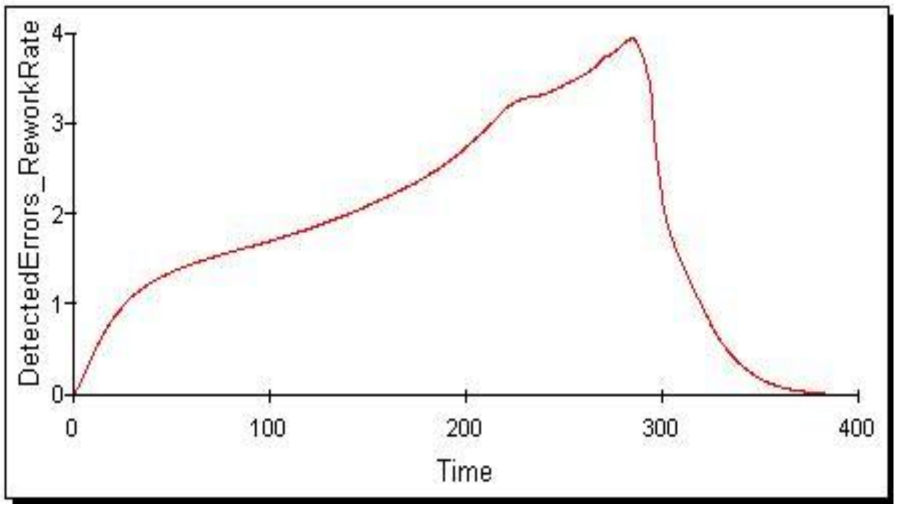


Figure 91. Detected errors rework rate

The second graph depicted here is the rework rate. As described in chapter 5 and 6 the rework effort is dependent on key factors during software production. These are similar to the factors influencing the detected errors waiting for rework rate. The factors are project progress, motivation, productivity, experiences of workforce, schedule pressure and manpower allocation.

As soon as errors are detected and need to be reworked, the correction-process starts. From the graph we see how the rework-rate increases sharply as errors are being corrected. The rate of correction starts out strongly as the project has recently been initialized. When a project is in the starting phase, the staff allocated is all rested and ready to work. Therefore the initial effort put into the earlier tasks are higher as motivation and optimism is high. With more errors detected the stock of errors needed to be reworked increases. Since the logic of positive motivation applies for rework as well, the team is initially reworking at a higher rate.

After a period of time the rework rate levels off. The reason for this lies in the effort and the exhaustion level. In the earliest stages of development, the tasks that are completed are design tasks. Design errors are more difficult to rework and the effort is high. After a few weeks of reworking difficult errors, the rework staff is getting less productive. The amount of errors corrected every day declines slightly, an indication of fatigue from the rework staff. Rework-teams are allocated from development, so every staff set to correct mistakes is unable to take part in the development of the project. This can lead to a negative association with rework, and lower overall productivity as well. The result is a lower rework rate.

A second factor explaining the decline here is manpower allocation for rework. At the beginning of the project when quality assurance detects errors and sends them to rework, the stock of detected errors waiting for rework, increases sharply. To accommodate this increase, manpower is allocated to rework in order to correct the waiting mistakes. When errors have been corrected at a strong rate, management is inclined to re-allocate some of the members back to production.

The third factor here is turnover. If there has been a cycle of turnovers after the initial start of the project, the rework staff has lost experienced members. Experienced members are valuable, as their expertise provide high productivity. Secondly, with turnovers the communication-overheads increase as well since the members of the team have new people to learn to communicate with. Continuous loss of partners in a working-team leads to motivational losses, as workers feel a sense of instability in the workplace.

After a while, the rework-rate picks up speed again. In the graph we see an in increase in the curve providing the second strongest rework-rate increase. The first explanation lies in the software production efficiency. When more tasks are being completed, the project moves from design to coding. In the coding phase, errors that need to be reworked are much simpler to handle. This reduces the rework effort needed per average error, and the same amount of staff can correct more tasks. In addition, when a project is moving from design to coding the staff has gained more knowledge of the inner workings of the program. This knowledge allows them to be more effective in correcting errors and the rework rate increases as a result.

The second explanation lies in manpower allocation. We saw that the quality assurance effort detected more errors towards the end of the project, increasing the stock of detected errors waiting to be reworked. The managerial response is more manpower for the rework effort.

Finally the amount of schedule pressures increases the rework rate for a period of time. As the project closes to the finish-line the schedule pressures increase. Increase schedule pressures provide a feeling of urgency in the rework team, and they will cut slack time and increase their productivity. This translates into a higher rate of rework as we can observe from the graph above.

The final pattern observed the reference mode is the decline of both the detected errors waiting for rework, and the rework rate. When 80% of all software tasks are completed, manpower is redistributed to the final systems test. The result is visible in the graph above. Since all production tasks have been completed, the stock of detected errors waiting for rework slowly declines. The rework-effort is still active, finishing the final correction-tasks. However, since manpower is re-routed to the final systems test the rework-staff slowly decreases over time as well. Hence, the rework-rate slowly decreases as the amount of errors waiting for rework reaches zero as more staff is moved from rework to testing.

The factors that govern the behavior of the system will have an impact on my policies as well. It is therefore very important that my analysis take them into consideration. For the final part of this chapter, I will provide a comprehensive analysis of the combined policies I synthesize through this chapter. For this analysis to be a contribution for managerial decisions, it must be clear to management the benefits and challenges that each policy address. Therefore they must be explained through the reference mode in order to hold any significant explanatory power.

8.1.1 Static Policy

The first policy area that I will analyze is the manpower allocated for rework. This is a variable that is controlled by a constant in the model, and therefore the policy constitutes what is known as a static policy. Static policies are policies that are governed by a pre-set fixed value through the entire modeled process. For example, the amount of time it takes for parcel to move from city A to city B is 3 days. This delay-time is constant, and businesses that send parcels between city A and B calculate this delay into their operation.

In some cases, the delay-time is set by management or by a controller as a desired time-frame. This will induce changes in policy and execution in a project depending on the time set. To exemplify I return to the previous example. The mayor of city A is concerned with the time it takes to send a parcel to city B. He wants the travel time for a parcel between city A and B to be 1 day rather than 3 days. He commissions a board of experts to provide a plan where the travel days for a parcel between A and B is reduced from 3 to 1. This reduction in desired time-delay will provide policies where for example the mode of transportation is changed from road-travel to air-travel. Other policies could include building new roads that are faster to travel on, or speed up the parcel-processing procedure. The governing variable is fixed, and therefore static. The policies are all constructed as a result of this pre-fixed and constant desire for a 1 day delay time.

This desired delay will not change during the course of the project, and the policy is therefore static. In my model, the allocation of rework manpower hinges on such a pre-set desired delay time. This is set during the planning-phase by the client and software management team.

8.1.1.1 Policy 1: Allocation of Manpower for rework effort

The first policy area to be analyzed is the rework effort. The rework effort is governed by the policy variable of the desired rework delay. This is a fixed time-period that dictates the time elapsed from the moment an error is identified by quality assurance, and the initialization of the rework effort to correct it. Since this is a pre-set time-delay, the policies deriving from the variable are static policies.

When errors are found by either quality assurance or the testing effort, the task is sent back for correction. This is carried out by the rework team. Rework-teams are allocated from the software production staff, and it falls on them to makes sure the end-product is defect-free. In the model, the allocation of manpower to the rework-effort is governed by the policy variable that is the “Desired Rework Delay”. This delay derives from the “Desired Error Correction Rate”. When an error is detected, it is usually not corrected immediately. When an error is reported, an amount of time elapses before a software-professional handles it. According to Boehms TRW data, this delay was found to be between 8-19 days (Boehm, 1981). Hence, the rework effort in strength depends on the time-delay from when an error occurs until it should be corrected.

A short desired rework delay provides more people allocated to rework as errors are found, but derives more effort from production. A long delay provides more tasks achieved, but more errors piled up for correction. Mismanaging this relationship could spell dire consequences for the effort as a whole. Over-allocate to rework, and the software project could be severely delayed. Under-allocate and the rework team could end up in a situation where they are swamped with errors at the final stages of the project causing severe delays and cost-overruns. (Abdel-Hamid, 1991: p. 75)

The normal managerial policy is provided by the base-model of Abdel-Hamid. The delay-time is set to 15 days, according to his own empirical data (Abdel-Hamid, 1991: p.75). With this policy it will take 15 days from an error has been detected before it is handled by a rework staff member assigned to the rework-detail. This variable is a constant, so for the entire production- cycle 15 days delay is set for all errors detected until they are reworked.

In policy 1 strategy 1 we find a more quality-oriented approach that diverts manpower from production to rework at an earlier time-frame. The desired delay time is therefore cut by 5% from the normal base-run. In this case, management is eager to see correction being performed. As a result the rework effort starts 10 days after the error has been detected. This allows the correction-effort to start earlier.

The perception from management here is a concern towards end-product quality. As soon as errors are removed and corrected, the software will benefit from increased performance. Early correction of errors will remove them from the system and make testing easier to perform.

The danger lies in the diversion of manpower from production to testing. Software manpower productivity is a source for increased performance in all efforts. The production of software facilitates learning and knowledge about its inner mechanics that is valuable for error correction as well. If management diverts manpower too soon for correction, their knowledge of the system is not as up-to-date as it could have been. This could result in a higher fraction of bad fixes that testing re-captures and sends back at a later stage. Balance is very important to ensure a profitable result for the project as a whole.

The policy 1 strategy 2 in this case, is a more production-oriented approach that focuses on task-production. In order to ensure a higher software production rate, the delay-time for an error is set to 5% longer than the base value. For this policy strategy the desired rework delay-time is 20 days for an error to be reworked after detection. So less manpower from production is allocated to rework effort. This allows more tasks to be produced over the same time-period as in strategy 1, as more tasks are completed before parts of the manpower is diverted to do rework. The benefit lies in increased production and production-efficiency, the danger lies in the accumulation of detected errors due to an increase tasks being completed. If the amount of errors detected rises above the potential rework-level, the rework effort will be swamped with errors to correct towards the end of the project. This is expressed in the stock of detected errors waiting for rework as an increase

In the two graphs below in figure 92, I provide the reference mode with the two policy strategies and the base-run. The first run is the normal base run depicted by the red line, while the green second run represents the behavior with strategy 1. The third blue line depicts the behavior of the key variables when strategy 2 is deployed.

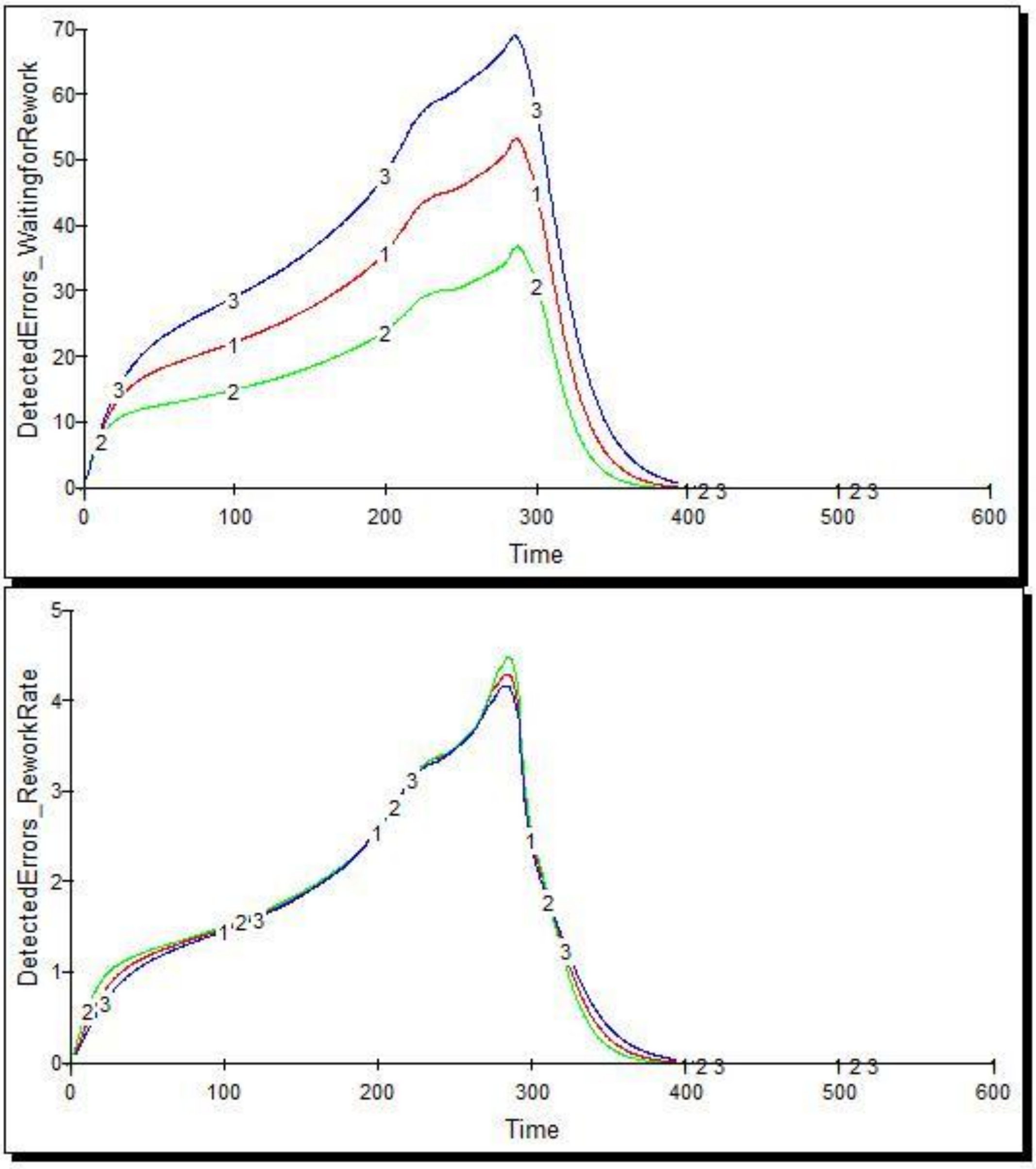


Figure 92. Detected errors waiting for rework and detected errors rework rate,
Red line presents the base run, green line presents the strategy1 and blue line presents strategy2

From the graphs it is apparent that the policy-strategies provide different outcomes in regards to both the number of detected errors waiting for rework, and the rework rate. The initial development in all three cases follow the same trajectory pattern, but the effect of the factors influencing the behavior is numerically different between the policies. For policy strategy 1, the shorter delay-time for the rework effort provides a larger rework staff. From the stock of detected errors waiting for rework, we observe that the amount is much lower than the reference mode.

There are two reasons for this development; firstly the larger rework staff is able to rework more errors at a shorter time-span. Secondly the increase in manpower for the rework effort reduces the amount of resources for software production. This translates into fewer tasks produced and less tasks for quality assurance to investigate. The increase in detected errors waiting for rework follows a lower and smoother trajectory compared to the reference mode. The positive effects from experience, learning and positive motivation are higher in the larger rework workforce team. In the same manner the negative effects of schedule pressure, loss of motivation and inexperienced staff are less since the workforce is more robust. This factor is evident in the second graph above in figure 92.

The rework rate for policy strategy 1 is higher than the base policy. The effort starts earlier, and is able to maintain a high level of rework-rate for a longer time-period than the base run. The negative effects of fatigue and motivational losses do decrease the rework rate for a period of time, but it maintains a higher level through the course of the software process. The positive effects of experience and learning, plus the effect of easier rework-tasks at the end of the project provides a peak in the rework-rate that is higher than base. In the end we also observe that the rework effort is able to finish the remaining errors at a higher rate after all tasks are finished.

For strategy 2 we observe that having a longer desired rework delay-time for the rework-effort causes the amount of errors waiting for rework to increase above the reference level. Due to a smaller rework staff, more resources are allocated to production. Increased production of tasks leads to more errors detected by the quality assurance team, and the amount of errors waiting for rework cumulates at a high rate.

The effects that influence the rework-effort such as fatigue, work-force mix, learning and experience are significantly different as well. With a smaller rework-team the negative effects of schedule pressures, stress and the workforce mix are higher. The positive effects from for example learning and experience is lower due to the smaller workforce. This effect is evident from the second graph, the rework rate. The rework rate for strategy 2 follows the same behavior pattern of the reference mode but at a lower level. The smaller staff is unable to obtain the same rework rate. The effects of fatigue and motivational losses are apparent as the rate levels off at a lower level than for the reference run.

More rework due to increased production tires the smaller rework staff, and they lose motivation. When errors have cumulated for a period of time, management adjusts the staff-size in order to accommodate the increase in detected errors waiting for rework. This gives the rework effort an increased boost in productivity, along with the positive effects of experience and learning. Additionally, coding errors are easier to rework than design errors. The negative effect of schedule pressure and fatigue levels increase in the rework effort towards the end of the project. At this point all tasks have been completed, and the rework teams are transferred to testing. The rework rate is at all times lower than the base run.

For strategy 2 we observe that the smaller rework staff is unable to correct the same amount of mistakes as strategy 1, and the base-run. This is depicted in the first graph, as the stock of detected errors waiting for rework increases to a higher level. For the rework rate we observe that a smaller staff is unable to be as effective as the base-model and strategy 1.The effects of negative factors such as motivational losses, fatigue and turnovers are higher for a smaller group. This follows the logic that a small group is more susceptible to negative influences compared to a larger group in the same situation. This is due to size differences. Larger groups are still able to perform better under bad conditions since they have more staffs to deploy. The result is a rework rate that follows the trajectory of base-run and policy strategy 1 but is unable to reach the same level through the entire development cycle.

<u>**Reported completion time, defect removal, man-day expenditure and benefit-cost ratio**</u>

The first table below presented under this headline is the reported completion time for each policy strategy. This table provides the exact number of days spent on the project, and shows how difference in allocation of manpower holds an impact on the completion time.

Policy simulation	**Completion time**
Base run	438 days
Policy 1 strategy 1	446 days
Policy 1 strategy 2	432 days

Figure 93. Project completion time for base run and policy strategies runs

From this table in figure 93, we observe the change in completion time when the distribution of manpower to the rework effort is changed through the desired rework delay. The original base-run completed the software project in 438 days. For strategy1, where the desired rework delay was cut to 10 days rather than the original 15 days, the completion time is longer. The project is finished after 446 days. This is due to the fact that more manpower was diverted from production to the rework effort. With more effort spent on the error correction effort, the software development rate was somewhat slower. The difference in completion-time is not a huge difference in itself. The completion-time is 8 days later for strategy 1.compared to the base run.

For strategy 2 the completion date is shorter than both base-run and policy strategy 1. This fact is not surprising due to two factors governing the completion time linked to the rework effort. The first factor is the allocation of manpower to rework that influences development productivity. For strategy 2 there is less manpower allocated for rework effort, and the desired rework delay-time is longer. As a result the manpower allocated for software development is higher for strategy 1 . Therefore more tasks are being completed at a higher rate.

The second factor is linked to the quality of rework. With more detected errors waiting for rework the schedule pressure rises and negative factors such as stress, fatigue and motivational losses impacts the correction effort. Through my previous chapters I have explained how bad fixes are influence by such conditions. When a small rework team is under pressure, the increase in error proneness leads to mistakes. More mistakes translate to a higher rate of bad fixes. According to Jones and Bonsignour, the level could be as high as 20% bad fixes for failed projects. Bad fixes leads to a situation where detected errors are reworked, but in an erroneous way. With less overall detected errors that are reworked, the effort is able to finish on time. However the amount of errors correctly removed is less. This phenomenon is observable in the next table reporting the total amount of errors removed from the software during production.

Policy simulation	**Defect removal**
Base run	709,40 errors
Policy 1 strategy 1	716,14 errors
Policy 1 strategy 2	703,82 errors

Figure 94. Project defect removal for base run and policy strategies runs

The table above reports the total number of errors that are detected, reworked and removed from the software during the production stage. The amounts of errors removed in a software project constitute the benefit of increased quality for the software project as a whole. For the base-run we observe that the total errors detected and removed was 709.40 errors. Compared with strategy 1 we observe that shortening the desired rework delay-time yield a positive increase in removed errors. Due to a larger staff and less delay, more detected errors are reworked and removed during production.

The reason for this increase compared with the base-run rests in experience, and bad fixes. Due to a shorter desired rework delay-time, the effort to remove errors starts earlier. This head-start allows the rework team to gain knowledge and experience of the nature of the defects. Since the effort starts sooner, the schedule pressure is lower which holds a positive effect on the ability to learn. Therefore the larger rework team is able to rework more errors, and avoid bad-fixes as a result of prior knowledge and experience. When schedule pressures do occur, the rework rate is affected. However, the larger team is able to work at a higher rate than in the base model. The experience from an early start allows them to avoid bad fixes at a higher degree as well.

For strategy 2 we observe the opposite phenomenon as explained above. The later initiation of the effort provides more schedule pressure for the rework-team at an earlier stage. Schedule pressures and stress do not facilitating learning. The size of the rework team is smaller as well, and the negative factors influence the group at a higher degree. The result is an increase in bad fixes. The result is a situation where fewer detected errors have been removed at the end of the project compared to both the base-run and strategy 1. When we compare the actual figures it is apparent that the difference between strategy 1, strategy 2 and base-run is not substantial. However, a few more errors slipping by in the development stage will lead to an increase in errors for testing, and errors that escape detection all together. Hence, software quality depends on the ability to find errors. Fewer errors removed equals lower software quality, and quality of the software development process.

The next report concerns the total man-days expended on the project. The total amount of man-days expended provides the total costs for the project. More man-days expended translate to higher costs due to longer production time and a larger amount of total workforce.

The table below provides the total man-days expended on the project under base run, policy1 strategy 1 and policy 1 strategy 2.

Policy simulation	**Man-day expenditure**
Base run	3878.70 Man-days
Policy 1 strategy 1	3919.91 Man-days
Policy 1 strategy 2	3845.66 Man-days

Figure 95. Project man-day expenditure for base run and policy strategies runs

In the original model, the project expended a total of 3878.70 Man-days. For strategy 1, we observe that deploying a shorter desired delay for the rework effort causes the amount of expended man-days to increase. Due to more manpower diverted from development to defect-removal, the project ran for a longer time-period. This increased the amount of man-days expended on the project as a whole. This result is in line with the previous reported results for the strategy 1. The difference in man-day expenditure between strategy 1 and the base-run is not extensive. However; there is an increase in costs by allocating more manpower to the rework-effort.

For strategy 2 the total amount of man-days expended are lower compared with the base-run and strategy 1. This is a result of the shorter completion-time for this strategy. Less allocation of manpower from production to rework leads to more allocation of manpower to development. This causes the software to be developed and completed at a higher rate. Hence, less man-days where expended on the project as a whole.

The next two graphs in figure 96 depict the development of detected errors reworked, and the man-days expended on the project during the software production cycle. The tables above provide the static end-result, while the graphs show the dynamic development over time for both factors. The first graph below depicts the cumulative detected errors reworked under policy 1 strategy 1 and strategy 2, compared with the base run. The second graph depicts the cumulative man-days expended on the project during development.

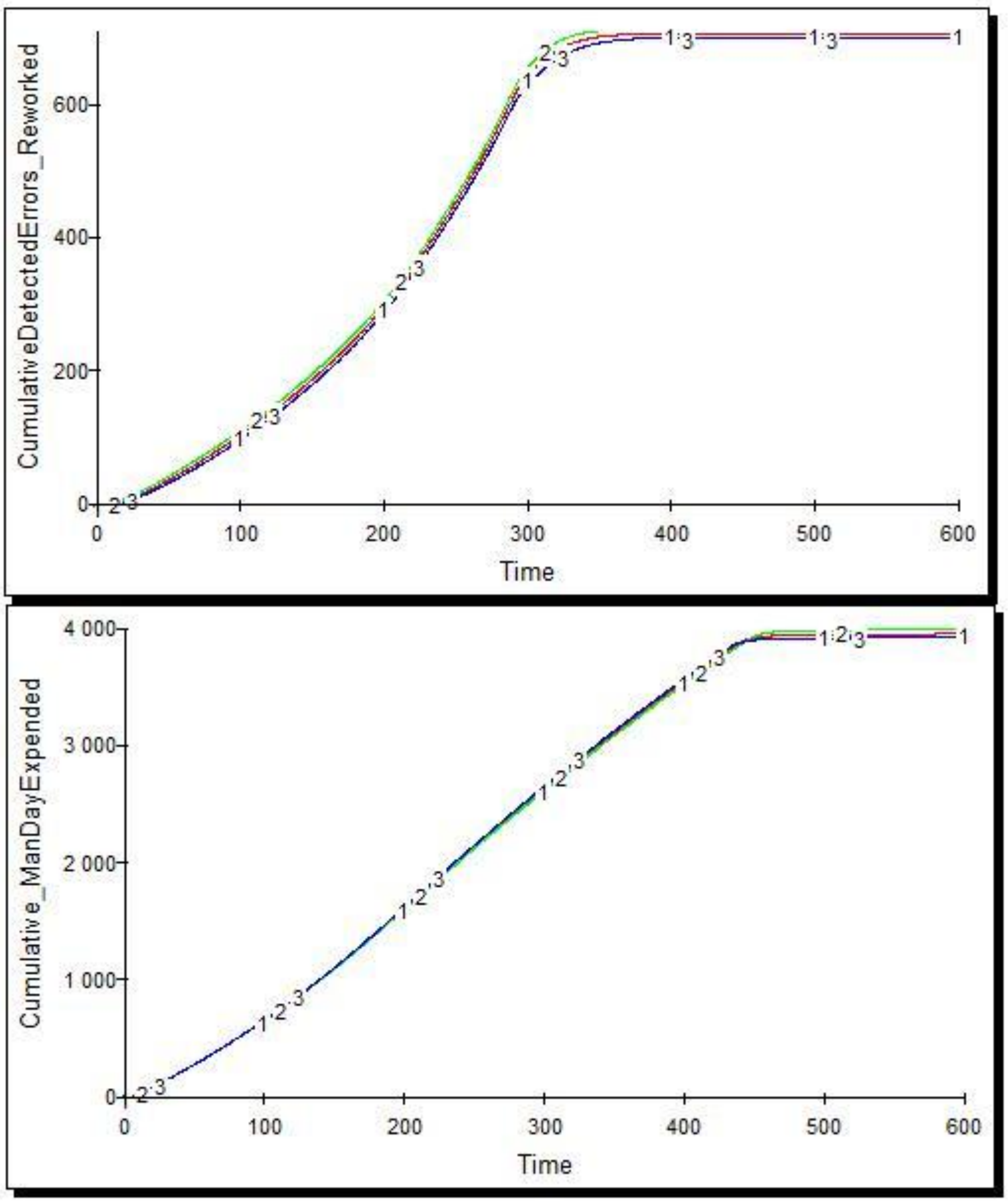


Figure 96. Cumulative detected errors reworked and cumulative man-day expeneded,
Red line presents the base run, green line presents the strategy1 and blue line presents strategy2

The development in the two graphs above in figure 96 is following an expected trajectory based on the reported end-results. Strategy 1 is able to start the rework-process earlier and the cumulative errors detected and reworked increases at an earlier stage. Compared to the base-run we observed that the defect removal rate was higher for strategy 1, and this difference is observable in the cumulative detected errors reworked. The trajectory of the increase matches the development in the base-run, but the level is higher until the end of the software project. The difference is not huge, something that was reflected in the reported completion time, total defect removal and man-days expended.

For strategy 2 the opposite result is observed. Due to a longer desired rework delay, the rework effort starts at a later time than the base run. Hence, the cumulative detected errors removed follow the same original trajectory-pattern. The level between the base-run and strategy 2 is different, and favors the base run. This is consistent with the final results reported in the tables above.

For man-day expenditures we observe that the pattern is a reversed image compared to the cumulative detected errors reworked rate between base-run, strategy 1 and strategy 2. Since the rework-effort starts early in strategy 1, the manpower diverted from development to rework increases. Hence, the project needs more time to finish all development tasks. The difference is not significant in the early stages of development, as man-days are expended at an equal rate between base-run and strategy 1. However, towards the end of the project the delay in completion time causes more man-days to be expended at a higher level for strategy 1 than for the base-run and strategy 2.

For strategy 2 man-days are expended at an equal rate compared to both base-run, and strategy 1. However, the difference is observable towards the end of the project. Due to a higher allocation to the development process, and fewer cumulative errors detected and reworked, the project completion time is shorter. The man-days expended at the end of the project are therefore less compared to strategy 1 and the base run.

When we observe the development in both graphs regarding the cumulative detected errors reworked, and the cumulative man-day expended, they display a characteristic pattern through the software development stage. This pattern is consistent with the base run, and follows the same logic. As the project is advancing, the defect removal effort starts to rework errors. As a result, the cumulative errors reworked increases. Due to factors like increased production, experience and progress from design to coding, the increase takes form of exponential growth. The same phenomenon occurs in the graph for man-days expended. The increase in activity increases the level of man-day expenditure in an exponential growth pattern.

After a period of time, the graphs change behavior-pattern towards the end of the project. There are fewer tasks that need to be completed, as the project is moving towards its completion. The goal for the project is to reach zero tasks remaining, and the behavior of the graphs depicts a goal-seeking behavior. The closer the project gets to the completion the graph levels off towards the cumulative end-level for both detected errors reworked and man-day expended. The result is a graph that follows an S-shaped development-pattern between initial exponential growth, and goal-seeking behavior. This is a characteristic behavior for these two graphs in this model. The behavior is the same between policy strategies and base-run, but the level is different due to policy strategies and manpower allocation.

When both benefit and costs have been analyzed for these two policy strategies, the final area to analyze is the benefit-cost ratio. This ratio is an expression of the difference in value between correcting and removing more errors versus the increased expenditure in man-days. For the policy to provide better software quality and at the same time be cost-efficient, the benefit-cost ratio must be higher than for the base-run. If the ratio is lower than the base-run it causes the increase in the total cost of the project. The final graph is depicted below in figure 97.

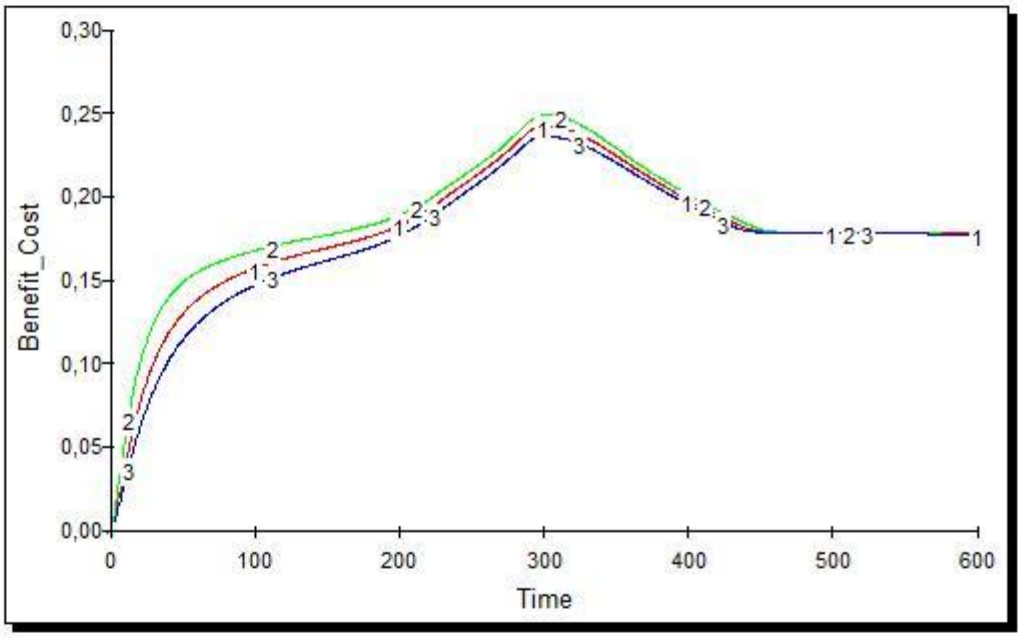


Figure 97. Benefit-cost ratio,
Red line presents the base run, green line presents the strategy1 and blue line presents strategy2

From this final graph we can analyze the benefit-cost ratio between policy 1 strategy 1 and strategy 2. The development observed in this graph is clear when it comes to the question of benefit versus cost. The decrease in the desired rework delay provides a higher rate of rework, and a larger rework staff. This effect is immediate, as the benefit-cost ratio increases above the base-run from the start of the project. The ability to remove errors early, to learn and gain experience, and achieve higher overall number corrected errors provides a higher benefit-cost ratio. The increase in man-days expended is not inflicting a higher cost compared to the benefit of the defect removal. The green line holds firm at a higher level than the base-run through the entire software process.

For policy 1 strategy 2, the opposite result is observed. A smaller rework staff and a longer desired rework delay cause the effort to be more costly than the base-run. Due to a smaller rework rate, and fewer errors removed overall the earlier completion-date is not causing a better benefit-cost ratio. The man-days expended are not yielding the same increase in software quality, and more bad fixes occur due to negative influences. Therefor the overall benefit-cost ratio for strategy 2 is less than the base-run.

Isolated, the policy strategy to encourage here is policy 1 strategy 1. However, there is a benefit in shorter development time. Secondly the amount of errors removed was not significantly larger between the two strategies. Hence, policy 1 strategy 2 has some positive sides that could be beneficial. This concludes the isolated analysis of policy 1, allocation of manpower to rework.

8.1.2 Dynamic Policy

Dynamic policies are policies that derive from variables and relationships that are susceptible to changes during the course of a project. The first policy I described in this chapter was a static policy as the delay-time set for rework maintains the same level through-out the entire software project. The next policies are all dynamic policies, as they are governed and adjusted as a result of the dynamic relationship and behavior of the model.

Dynamic policies are directly influenced by conditions inside the system dynamics model. These conditions can change over time, leading to an adjusted value for the variable that the policy derives from. For example will the level of city public transport maintenance for the mayor in city A change with the seasons of a year. During the winter-months more manpower and budget-money must be spent on snow-clearing in order to have airports, road-networks and railway-tracks operational than in the warmer summer months. This relationship makes it impossible to have a static budget allocation for such activities during the entire year. Hence the policy is sensitive to dynamic changes in conditions (in this case weather and temperature).

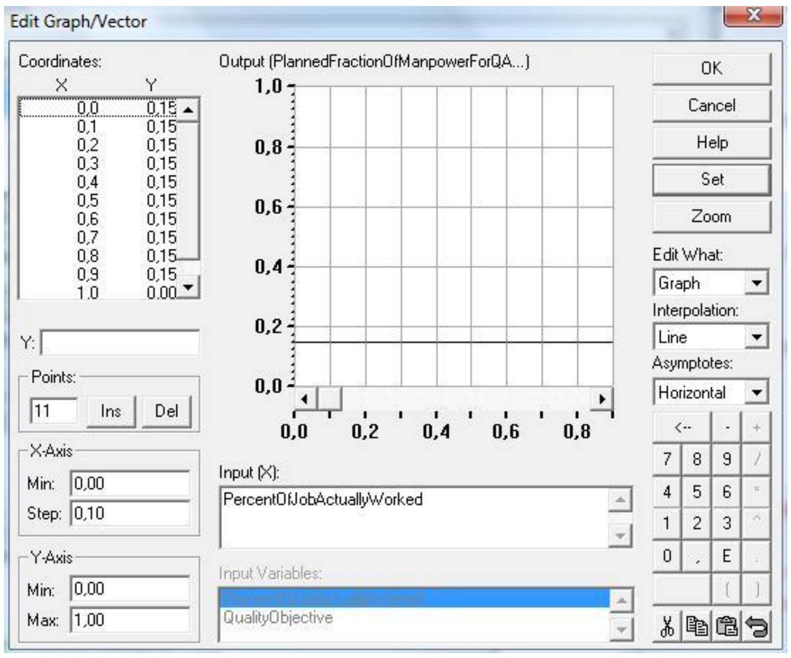
The next policies that are analyzed in this chapter are all dynamic policies. The prime variables that dictate the policy is dependent on the conditions within the software project, and its progress.

8.1.2.1 Policy 2: Allocation of manpower for quality assurance effort

The next policy area that will be analyzed is the manpower allocated to the quality assurance effort. In this model, manpower allocation is controlled trough the fraction called "Planned Fraction Of Manpower For quality assurance". As the name indicates, the allocation of manpower follows a planned fraction set by management at the beginning of a software project. It is necessary to decide on an allocation-level for this effort, in order to make the first division of labor between the members of the workforce.

Unlike the rework-teams and testing-teams, the quality assurance staff is not linked to software production. The quality assurance staff is moved to the defect removal side of the project, and performs these tasks during the production cycle. The quality assurance teams provide both the bureaucratic functions of information-storage and project documentation, and the defect removal function. The latter function is achieved through inspections; walkthroughs, small scale integration-testing plus the capture re-capture effort. It is therefore important for management to allocate enough manpower for this effort, without compromising the software production side of the project. This poses a challenge for management since the quality assurance effort yield valuable information and ensures quality, but reduces the possible staff for the actual software production. If this balance is mismanaged, the results could be delays, cost-overruns, or faulty software. There are two possible policy-strategies for management, (1) a production-focused policy where management allocates more directly to production and (2) a defect-removal focused approach where more staff is allocated to quality assurance.

The policy variable in these strategies is the variable named “Planned Fraction Of Manpower For QA”. The graphs depicted below in figure 98 consist of the variable “Percent Of Job Actually Worked” on the X-axis and the fraction of planned manpower for QA on the Y axis.


Edit Graph/Vector

Coordinates:

X	Y
0,0	0,15
0,1	0,15
0,2	0,15
0,3	0,15
0,4	0,15
0,5	0,15
0,6	0,15
0,7	0,15
0,8	0,15
0,9	0,15
1,0	0,00

Output (PlannedFractionOfManpowerForQA...)

Y:

Points: 11

X-Axis: Min: 0,00 Step: 0,10

Y-Axis: Min: 0,00 Max: 1,00

Input (X): PercentOfJobActuallyWorked

Input Variables: QualityObjective

Edit What: Graph

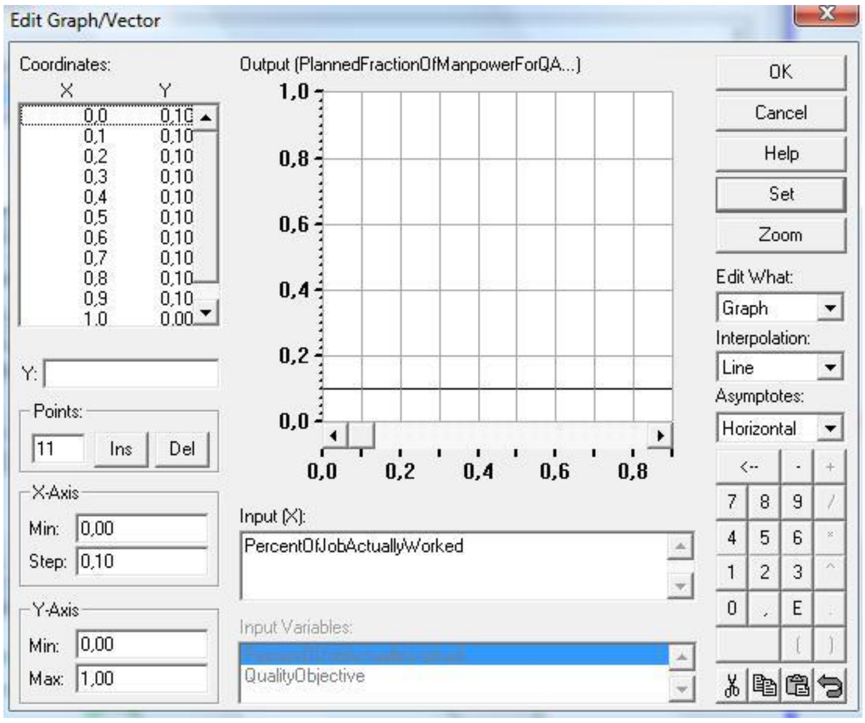
Interpolation: Line

Asymptotes: Horizontal

Figure 98. Planned fraction of manpower for quality assurance, Normal run

The base run is the normal decision that was presented by Abdel-Hamid in the original model. From the graph we see that the allocation of manpower to the quality assurance staff is 15% of total daily manpower after training. Through the entire project, from design to coding, the amount of manpower allocated to the quality assurance effort is 15% of the entire workforce after training and it drops to zero at the end of the project when all staffs have moved to final system testing. This assures a continuous quality assurance effort, where a fraction of new hires will be allocated to quality assurance if the project’s workforce increases in size. With an increased workforce, the number of tasks accomplished will increase. Therefore it is important for the quality assurance effort to absorb such variations.

As long as there are tasks being produced and the software project is progressing, staff is allocated to the quality assurance effort. However, as soon as all tasks have been completed and the project is finished, the quality assurance staff will be transferred to the final systems-test. This is expressed in the graph above, as the percent of job actually worked reach 1.0 the fraction of manpower allocated for quality assurance is set to zero.



Coordinates:

X	Y
0,0	0,10
0,1	0,10
0,2	0,10
0,3	0,10
0,4	0,10
0,5	0,10
0,6	0,10
0,7	0,10
0,8	0,10
0,9	0,10
1,0	0,00

Y:

Points: 11

X-Axis Min: 0,00 Step: 0,10

Y-Axis Min: 0,00 Max: 1,00

Output (PlannedFractionOfManpowerForQA...)

Input (X): PercentOfJobActuallyWorked

Input Variables: PercentOfJobActuallyWorked QualityObjective

Edit What: Graph

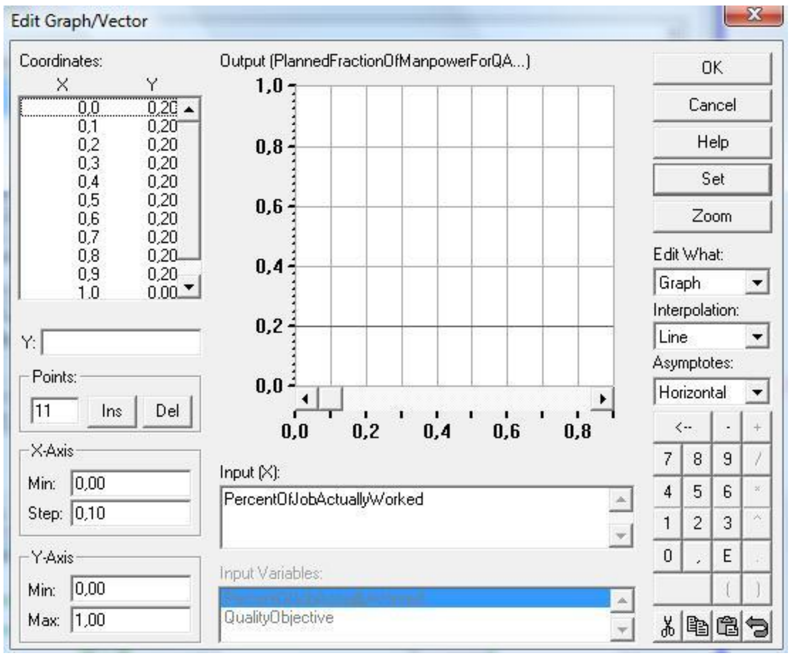
Interpolation: Line

Asymptotes: Horizontal

Figure 99. Planned fraction of manpower for quality assurance, Policy 2 strategy1

As shown in figure 99, the policy 2 strategy 1 presented is a production-oriented approach. In this policy-regime management's chief concern rests with production and the ability to finish software tasks as soon as possible. Since quality assurance staff is delegated away from production, the fraction of manpower is cut with 5% from the normal base-run. During the entire production-cycle, 10% of the total daily manpower after training will be allocated to quality assurance and the effort remains at 10% as long as there are tasks being accomplished. Any variation in total workforce-size ensures increases or decreases of manpower to the effort, to accommodate the 10% goal. As observed in the policy 2 strategy 1, when all tasks are accomplished the staff is transferred to the systems-test.

This form of allocation is found in typically in projects of a tighter schedule, or simpler types of software. According to Boehm, quality assurance provides both valuable information but also delay. (Boehm, 1981) Time spent in communication between members of production-staff and quality assurance leads to overheads. Such delays could push a project off schedule quickly if there is a tight deadline. This is a decision that is running a fair risk between completion and quality, as the end-product could end up as flawed software due to the reduction of quality assurance, and more escaped errors passing through the filter.



Output (PlannedFractionOfManpowerForQA...)

Coordinates:

X	Y
0,0	0,20
0,1	0,20
0,2	0,20
0,3	0,20
0,4	0,20
0,5	0,20
0,6	0,20
0,7	0,20
0,8	0,20
0,9	0,20
1,0	0,00

Y:

Points: 11

X-Axis
Min: 0,00
Step: 0,10

Y-Axis
Min: 0,00
Max: 1,00

Input (X): PercentOfJobActuallyWorked

Input Variables:
QualityObjective

Edit What: Graph
Interpolation: Line
Asymptotes: Horizontal

Figure 100. Planned fraction of manpower for quality assurance, Policy 2 strategy2

As shown in figure 100, policy 2 strategy 2 is a more quality-oriented strategy, where a higher percentage of staff allocated to quality assurance is pursued. In this strategy, management's chief concern is to ensure as sufficient a defect-removal effort as possible. In such a situation the allocation of staff to quality assurance is increased by 5% from the base-allocation provided in the original model. From the graph we see that the allocation of manpower is constantly held at 20% of total daily manpower after training as long as there are tasks being produced. This allocation ensures that variations in the total workforce will at any time during software production yield a 20% of staff to the quality assurance effort. At the end of the software project, staff is moved from quality assurance to the testing effort.

Under such a policy strategy, the chief concern for management is end-product quality. To avoid the risk of creating software with significant flaws, the quality assurance team is held at a higher level to pick up undetected errors. The crew is involved in detection through the inspections and the capture-recapture system. When the team is allocated at this rate, more errors will be detected at the first stage. The managerial decision to allocate more for quality assurance on the expense of production rests on a perception of early detection to avoid later delays.

For management, it is vital to balance production and defect-removal. Having a larger quality assurance allocation leads to less production, so it is risky in terms of delays. On the other hand, if the end-product is full of significant errors for the final systems test due to lack of quality assurance, the time saved by cutting this effort is gobbled up. In testing, an error is as much as ten times more expensive to correct and could lead the entire project into failure. Hence mismanaging the level of allocation in either way, the project could suffer both delays and cost-overruns. In the graphs below in figure 101, the two policy strategies are analyzed and compared with the base model.

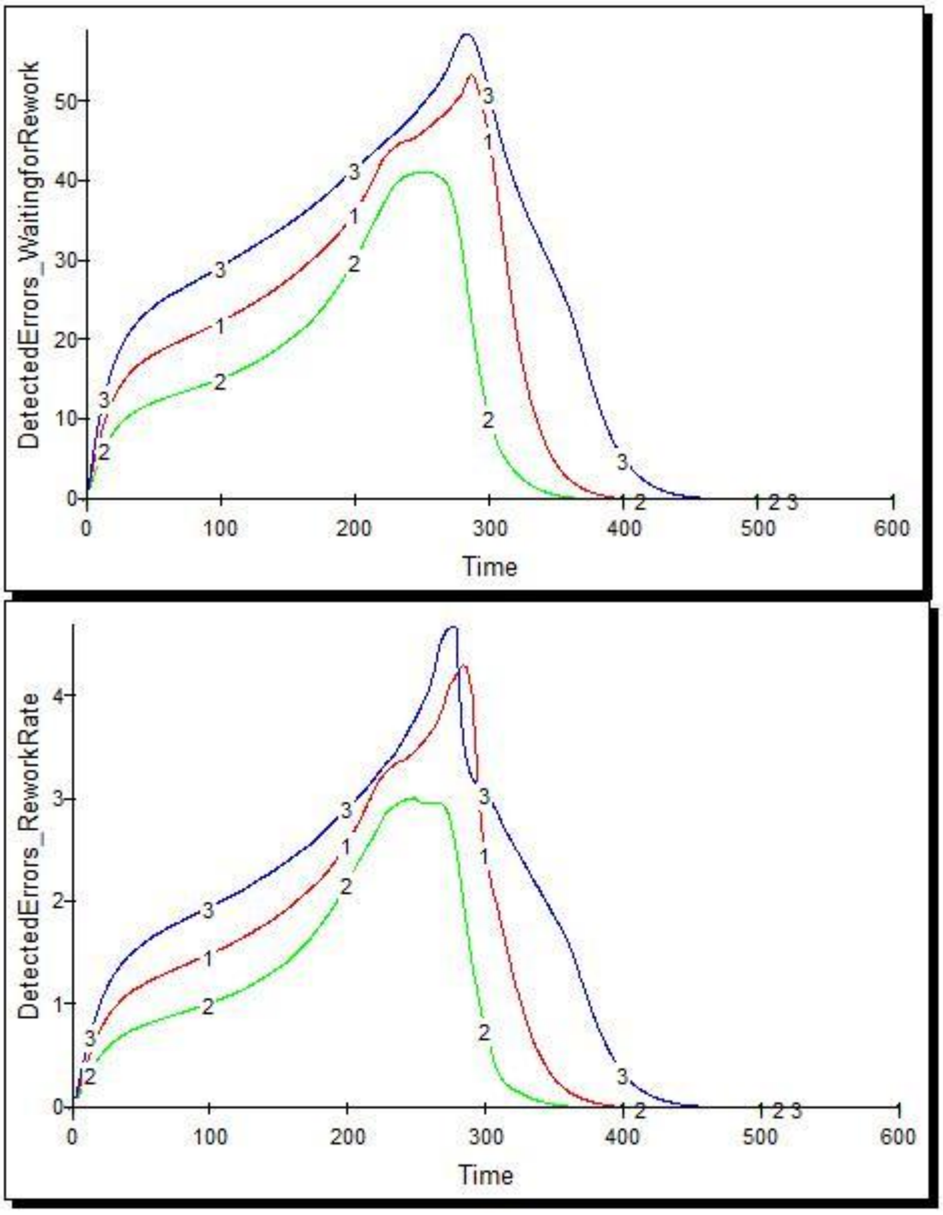


Figure 101. Detected errors waiting for rework and detected errors rework rate, Red line presents the base run, green line presents the strategy1 and blue line presents strategy2

From the two graphs above it is apparent that the level of manpower allocated to the quality assurance effort holds great influence on the detected errors waiting for rework, and the rework rate. For strategy 1, the allocation of manpower is reduced to a level lower than the base run. From the start we observe that the number of errors that are detected and waiting for rework increases at a lower rate than with a less amount of quality assurance staff. The ability to detect errors hinges on the quality assurance staff. The potential to detect errors is linked to the manpower allocated for this effort. When less manpower is allocated, the potential for error detection is lower. Hence, more errors will slip by as undetected errors and end up in the testing effort.

However, we observe that the stock of errors detected by this allocated effort increases over time. This increase is due to learning, experience, and the fact that more tasks are being produced. When less manpower is allocated for quality assurance, more manpower is dedicated to software production. This increase in completed tasks leads to a higher potential for detecting errors due to the task-amount. However, with a smaller staff the error detection potential is lower than the base run due to the fact that parts of quality assurance are linked to the tagging effort in the "capture-recapture" effort. In the end of the project the amount of errors detected by quality assurance increased to reach its peak but it is significantly lower than for the base-model. The increase in detected errors levels off towards the end of the project and the detection of errors end at an earlier time. This is due to the fact that increased production finishes all tasks at an earlier stage, and quality assurance manpower is moved to the final systems test.

The quality assurance staff holds a significant effect on the rework effort, as observed in graph number two in figure 101. Since fewer errors are detected in the initial phase of the software project, the allocation for the rework-effort is less. Management seeks to re-allocate as much manpower for the software production effort as possible, and few errors waiting for detection signals the opportunity for reallocation. This causes the rework rate to be lower during the entire software project cycle. The rework effort levels off at a significantly lower rate as a result.

Policy 2 strategy 2 employs a larger quality assurance allocation than the base run. With an increased allocation of manpower to the quality assurance effort, the potential for detecting errors is higher. This is depicted in the first graph in figure 101, as the increase in detected errors waiting for rework is significantly higher than initially. We observe that there is a stable increase in the detected errors waiting for rework, as the development and the quality assurance team gains experience and work more effectively. Towards the end of the project, an increase in the detected errors rate occurs, as the project moves from design to coding. This is due to the fact that more tasks are being completed by an increase in productivity due to learning and experiences for both development and quality assurances staff. In addition, the coding tasks are easier to develop and the coding errors easier to detect.

The detected errors rework rate increases significantly toward the end of the project. This is due to the relationships among rework, capture-recapture and testing. The rework rate is also highly sensitive to the allocation of manpower to the quality assurance process. When more errors are detected and waiting for rework, it gives a signal to management that an increase in the rework effort is needed. Hence, the rework rate holds a continuous higher rate than the base run. We also observe that the peak in the rework rate occurs earlier at a higher level.

At the end of the project, all the tasks have been completed. Manpower is moved from development, quality assurance and rework to testing. There are still tasks that contain errors and needs to be reworked. This effort is done by the remaining rework personnel, which are incrementally transferred to testing. The result is a behavior pattern in detected errors rework rate and detected errors waiting for rework that decreases in a smoothing fashion towards zero.

When we compare the development in both the stock of detected errors waiting for rework and the rework rate, it is apparent that the increase in quality assurance personnel is significant. The level of detected errors increase from the start, and the rework rate follows suit. Hence, more errors are detected in the early design-phase and the rework rate is increased to accommodate this increase in error detection. The potential to find errors depend on the quality assurance team’s size. This factor is depicted clearly in the detected errors graph.

Analysis report for completion time, detected errors reworked, man-days expended and benefit-cost ratio

The first table in Figure 102 that I present in this analysis is the reported completion time. One of the important aspects for a software project is the total time spent on production. The original base-run was able to complete all tasks in 438 days. In my isolated analysis, the level of quality assurance manpower was altered for the two policy strategies. The table below presents the completion time.

Policy simulation	Completion time
Base run	438 days
Policy 2 strategy 1	430.5 days
Policy 2 strategy 2	472.5 days

Figure 102. Project completion time for base run and policy strategies runs

In strategy 1 the quality assurance manpower was reduced to 10% of all available daily manpower after training. The result of reducing the number of quality assurance personnel yielded a shortening of the completion time. The project was able to finish 8 days earlier than for the base-run. This is a low and insignificant difference, and indicates that moving more manpower from quality assurance to software development is not an efficient strategy in terms of completion-time.

For strategy 2, the result is a longer development-time. The project is finished after 472 days, which is longer than the base run. Due to more errors being detected at an early stage, the rework effort has to increase their rework rate to accommodate the increase in detected errors waiting for rework. More manpower is initially diverted from development to defect-removal trough the increased fraction of manpower to quality assurance. In addition, the increase in detected errors signals for management that an increase in the rework-effort is necessary. Hence more overall manpower is diverted to defect-removal and the project runs for a longer time-period.

The second report in figure 103 concerns the total amount of errors that have been detected and corrected trough the software production cycle. For my policies to yield a better software quality product, the total number of errors removed from the software must be higher than the original base run. The table below contains the total amount of errors detected and removed. This factor is the benefit side of the project.

Policy simulation	Defect removal
Base run	709,40 errors
Policy 2 strategy 1	481.29 errors
Policy 2 strategy2	926.21 errors

Figure 103. Project defect removal for base run and policy strategies runs

The original base-model was able to remove 709.40 errors during the software development cycle. For strategy 1 it is apparent from the table that reducing the quality assurance effort is a poor decision regarding the ability to find and remove errors. The drop in detected errors falls by over 200 errors, which is a significant difference. The potential for error detection is directly determined by the size of the quality assurance effort, as described by Abdel-Hamid (Abdel-Hamid, 1991: p). By reducing the quality assurance staff with 5% from the base model, the ability to detect errors falls dramatically. For a project, the deployment of strategy 1 would lead to a situation where an additional 228 errors would stream through production undetected without a significantly shorter completion time. This would cause a significant drop in software quality, and hamper the quality of process as these errors would cause increased effort for the final systems test.

For strategy 2 the increase in quality assurance yields an equal positive effect. By increasing the size of the quality assurance staff to 20% of all available daily manpower after training, the potential for error detection increases drastically. By the end of the software project, the defect-removal effort has corrected 926.21 errors. This is an increase in detected errors by 217. This is a significant improvement compared to the base-run. This table supports the empirical literature on the field that highlights the importance of quality assurance as an effective defect removal tool (Jones & Bonsignour, 2012).

The final table in figure 104 presented in this analysis, is the cost side of the project. Project costs are measured in man-days expended on the project. Hence, the higher man-day expenditure, the higher the total costs for the project as a whole.

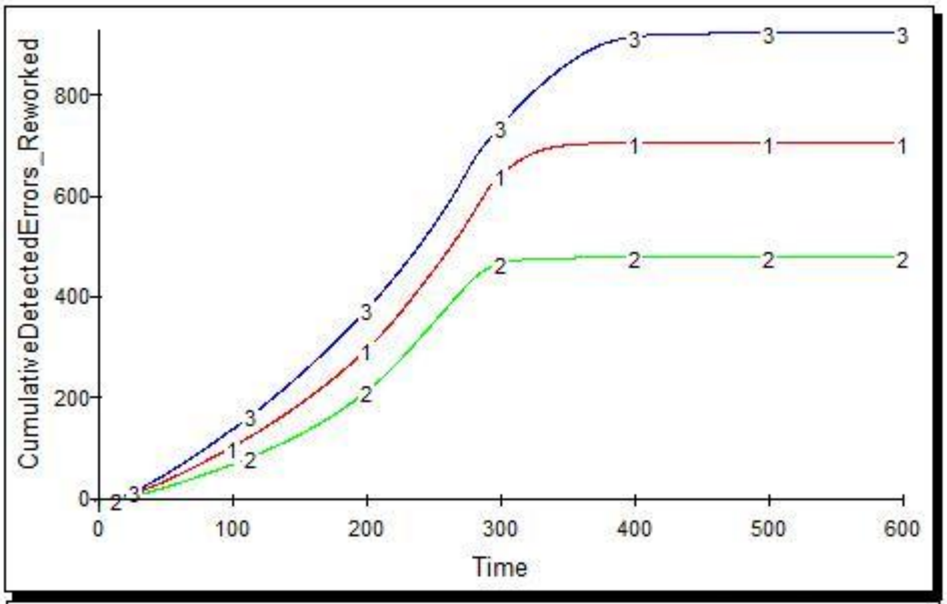
Policy simulation	**Man-day expenditure**
Base run	3878,7 Man-days
Policy 2 strategy 1	4021,45 Man-days
Policy 2 strategy 2	4055,47 Man-days

Figure 104. Project man-day expenditure for base run and policy strategies runs

For policy strategy 1, the decision to reduce the fraction of quality assurance personnel yields an increase in the total man-days expended on the project. The reduction causes a loss for the potential to detect and remove errors, at the same time as the man-day expenditure is higher than for the base-run. When less manpower is allocated to the quality assurance process, the potential detection rate decreases. When the potential detection rate is low, the amount of errors that slip through and escaped detection increases. These errors will remain in the software until the final systems test. Escaped errors increase the testing-effort needed per average error significantly. Hence, lower testing rate causes a slower testing-effort. More man-days are expended due to this delay. Errors that are discovered in a final systems test are more costly and difficult to remove. Therefore the man-day expenditure is higher for policy strategy 1 compared to the base-model even though the project is finished earlier due to the shorter completion time.

The result from strategy 2 is as expected from the analysis of the detected errors waiting for rework, and the rework rate. Large QA staff leads to more errors detected, and the rework rate is increased. This causes a higher effort to defect removal, lowering the potential development efficiency. The project runs longer than the base-model and more man-days are expended as a result. The differences in man-days are not dramatic, but the overall cost to the project is higher compared to the original model. This concludes my table-presentation for this isolated analysis.

As shown in Figure 105, the next part of my analysis is to present the graphs depicting the cumulative detected errors reworked, and the cumulative man-days expended. The tables report the static final results for the benefit and cost element. The graphs on the other hand, depict the dynamic development of the cumulative stocks as the development process progresses. Hence, we can determine at what specific time the impacts of difference in the policy-strategies cause the reported end-result.



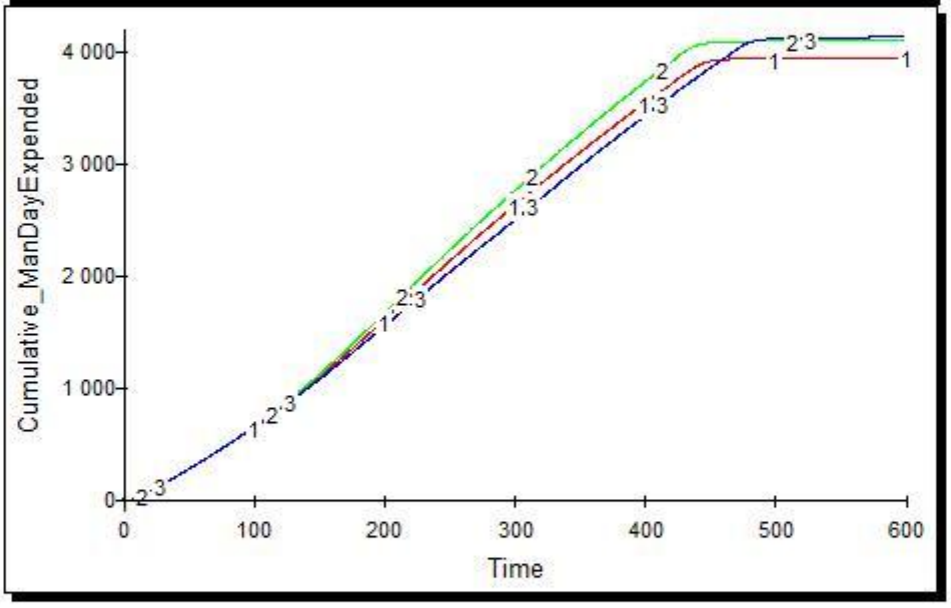


Figure 105. Cumulative detected errors reworked and cumulative man-day expeneded, Red line presents the base run, green line presents the strategy1 and blue line presents strategy2

The first graph in figure 105 depicts the cumulative detected and reworked errors for strategy 1 and strategy 2 together with the base run. For strategy 1 it is clear that reducing the quality assurance effort yields a loss in detected errors from the start of the project. The total amount of reworked errors holds a significantly lower level through the entire production-cycle compared to the base run. The trajectory of the development follows the same pattern as for the base-run, but levels off at an earlier stage. The level-off point is significantly lower than in the base run, which is consistent with the reported total of corrected errors.

Strategy 2 provides a clear image for increase in defect removal with a higher fraction of quality assurance staff. In the critical design-stages, the error density is low. This requires a higher quality assurance effort to detect these well hidden errors. The effect of having a larger staff that increases the potential error detection rate increases the amount of corrected errors beyond the level of the base-model. The trajectory and behavior of the graph mimics the base-run, but the level of defects removed from the software is continuously higher. This reflects the reported total of detected and removed errors from figure 103.

For the man-day expenditure there are some interesting to observations between the policy-strategies and the base-run. For strategy 2 that incorporates a higher quality assurance staff, the initial man-day expenditure is lower than for the base-model. The early detection and rework-effort allows both defect-removal efforts to gain valuable experience and knowledge at an earlier stage. This factor allows the defect removal to occur at a higher rate through the entire software development phase. This causes the expended man-days needed to be lower than the base-run until the very end of the project. The increased amount of time needed to rework the extra level of detected errors, causes the project to run longer. Hence, more man-days are expended to accommodate this level at the final stages of the project.

The two graphs above display the characteristic behavior explained earlier in this chapter. The initial behavior for cumulative detected errors reworked, and man-day expended is an increase as activity in the project initiates. The result is an initial behavior of exponential growth. However, as the project moves towards completion the behavior pattern changes. The goal for the project is zero remaining tasks. When the project progresses towards this goal, the level of detected errors reworked and man-day expenditure levels off towards the goal. The graph displays an S-shape behavior-pattern as a result. This observed development for both strategies 1 and 2 is consistent with the base-run but at a different numerical level.

The two graphs above depict the benefits and costs of both policy strategies. The final graph I will present in this analysis in figure 106 depicts the benefit-cost ratio. It is important that the policy strategy chosen for the project ensures product and process quality, without increasing the costs to a non-beneficial level. Like in the previous analysis, the graph depicts both policy strategies 1 and 2. These are comparable to the base-run depicted as the red line with the “1” brand. For a policy to be a viable strategy, the benefit-cost ratio must be higher than for the base run.

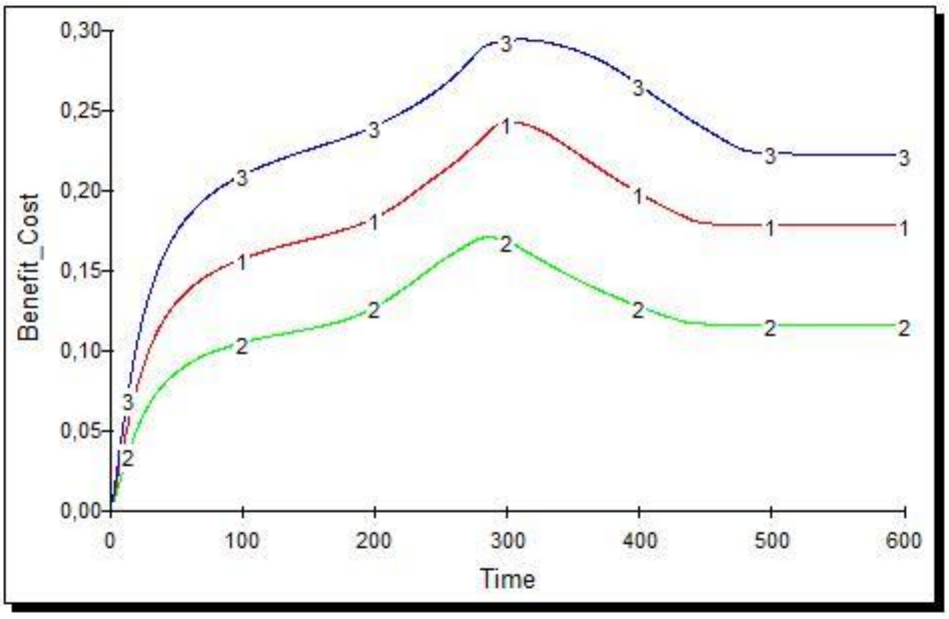


Figure 106. Benefit-cost,
Red line presents the base run, green line presents the strategy1 and blue line presents strategy2

Not surprisingly, the benefit-cost ratio for strategy 2 is much higher than in the base run. With this strategy, the quality assurance effort detected a significantly larger number of errors that were removed from the end product. The difference in man-days was not large, and therefore the benefit-cost ratio depicted above reflects a stronger benefit rate. The quality assurance effort was able to detect more errors at an early stage, and therefore the benefit surpasses the base-run from the initiation of the project. This higher level is maintained through the entire software development cycle. The observed increases and decreases in the benefit-cost ratio is caused by the positive and negative influences explained in detail in the first part of this isolated analysis. Productivity, experience, positive motivation and progress yield higher efficiency and better benefit-cost ratio. Overheads, turnovers, schedule pressure, bad fixes and loss of motivation reduce efficiency and lower the benefit-cost ratio.

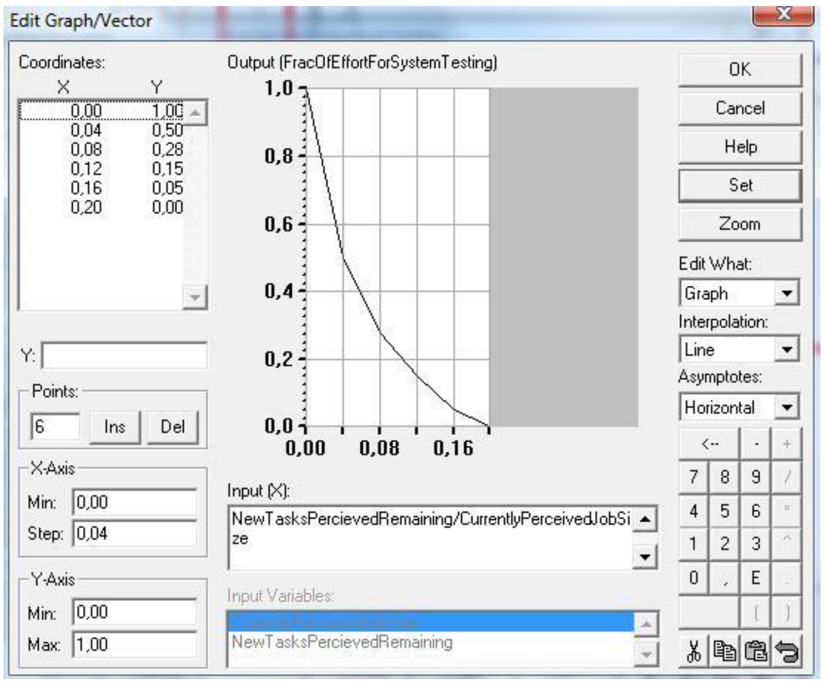
For strategy 1 the reduction in quality assurance manpower yields a worse benefit-cost ratio as compared to the base run. For this strategy, the amount of errors detected was significantly lower than in the base-run. The completion-date was not significantly different, and the man-days expended on the project surpassed the level of the base-run. Hence, strategy 2 is not a beneficial strategy for either software quality or project costs. This concludes the isolated analysis of policy 2 concerning manpower-allocation for the quality assurance process.

8.1.2.2 Policy 3: Allocation of manpower for testing effort

The policy variable of the "Fraction Of Effort For System Testing" effects the manpower allocation for the testing process through change in allocation policy. In the base model Abdel-Hamid assumed that during the project, people from the development effort would be reassigned to testing as the project moved towards completion. The daily manpower for the software development effort contains the potential workforce for the testing phase. This is due to the fact that testing-personnel are designated from the software production staff. In the base model, the allocation of manpower to testing was done at the very end of a project. The variable Fraction of Effort for System Testing describes this fact. The value for this variable is initially zero since no effort is initially allocated for System Testing. When development is perceived to be completed, this value becomes with a certain overlap (Abdel-Hamid, 1991: p.79, p.115).

The reason for this is the assumption that a software project only holds a final system testing. When the model of Abdel-Hamid was published in 1991, the software applied was simpler than the software created today. During the computer-revolution in the 90s and 2000nds, software has become far more complex. In the literature it is evident that testing as an effort has grown in scope and practice over the last two decades. In a modern software project, testing is carried out at an earlier stage and provides valuable information and milestones for the project. This is due to new technology that makes it possible and feasible cost-wise. Therefore, as discussed in detail in chapter 6, this assumption is changed in my enhanced model. Testing will be carried out in several stages during development process that includes both design and coding phases. The testing effort is now providing integration tests and component tests of finished parts of the software. In a medium ranged project, the software will have a multitude of function that together adds up to the end-product. Testing will commence as soon as the first design-aspects have been completed and components can be tested (Jones & Bonsignour, 2012).

The policy variable here is the variable named “Fraction Of Effort For System Testing”. The graphs depicted below in figure 107 consist of the variable “New Tasks Perceived Remaining / Currently Perceived Job Size” on the X-axis and the “Fraction of Effort for System Testing” on the Y axis. In the next part, I will present this relationship in base model and under different managerial decisions.



Edit Graph/Vector

Coordinates:

X	Y
0,00	1,00
0,04	0,50
0,08	0,28
0,12	0,15
0,16	0,05
0,20	0,00

Output (FracOfEffortForSystemTesting)

Y:

Points: 6 Ins Del

X-Axis Min: 0,00 Step: 0,04

Y-Axis Min: 0,00 Max: 1,00

Input (X): NewTasksPercievedRemaining/CurrentlyPerceivedJobSize

Input Variables: NewTasksPercievedRemaining

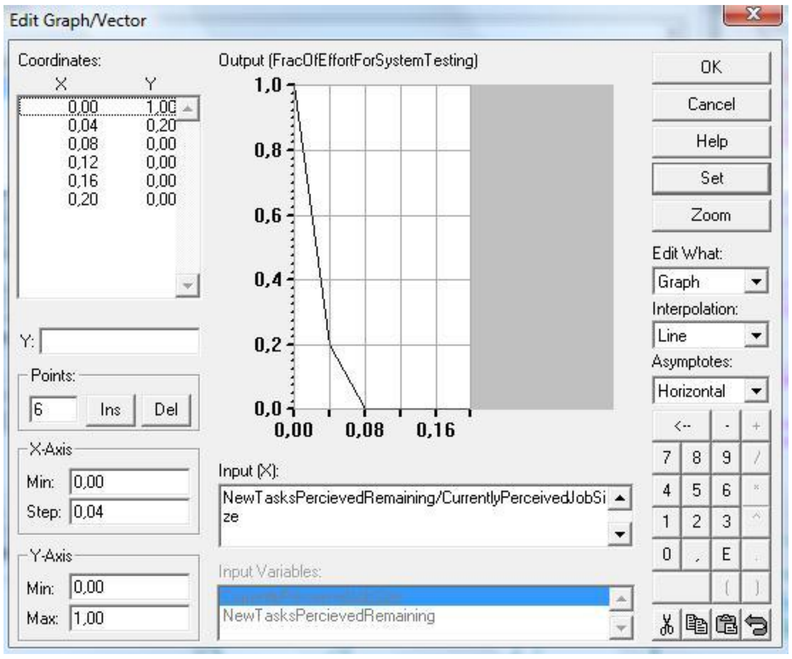
Edit What: Graph

Interpolation: Line

Asymptotes: Horizontal

Figure 107. Fraction of effort for system testing,
Normal run

The graph depicted above in figure 107 is the normal base-strategy of the model of Abdel-Hamid. In the graph, the x-axis represents the number of “New tasks perceived remaining” divided by the “Currently perceived job-size”. This policy variable holds a value of 1 only when no new tasks are perceived still remained. As described by the variable, manpower to testing will be allocated near to the end of the project. From the graph we see that allocation increases significantly at the second half of coding phase, when 80% of tasks are completed. When several new tasks are perceived still remaining, the allocation to the testing effort is non-existent or very low. This is a testing-regime tailored for one single test, where the focus on software production is higher. In addition, the quality assurance effort and rework are perceived to be the prioritized effort for defect removal.



Edit Graph/Vector

Coordinates:

X	Y
0,00	1,00
0,04	0,20
0,08	0,00
0,12	0,00
0,16	0,00
0,20	0,00

Output (FracOfEffortForSystemTesting)

Points: 6

X-Axis Min: 0,00 Step: 0,04

Y-Axis Min: 0,00 Max: 1,00

Input (X): NewTasksPercievedRemaining/CurrentlyPerceivedJobSize

Input Variables: NewTasksPercievedRemaining

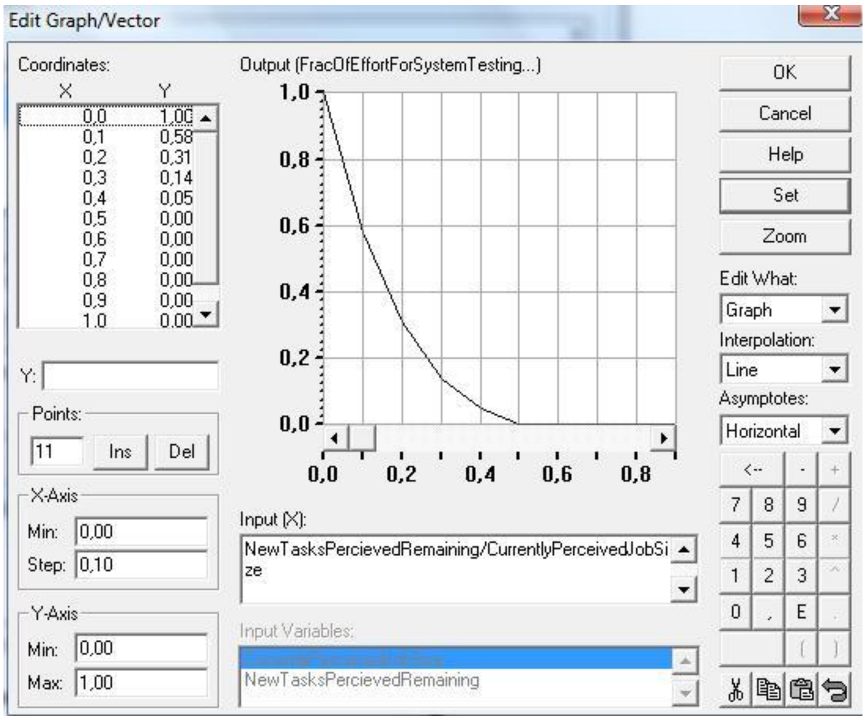
Edit What: Graph

Interpolation: Line

Asymptotes: Horizontal

Figure 108. Fraction of effort for system testing, Policy 3 strategy1

The policy 3 strategy 1 deployed here in Figure 108 is an even more aggressive approach to testing than the previous normal approach. Management in this circumstance is favoring software production over testing. The allocation to testing will not be activated until 90% of all tasks are done. From the graph above we can see the sharper angle of the line, as no overlap appears in the earlier stages. The graph holds a value of 0 until only 10% of new tasks are still perceived to remain. Then management allocates at a high tempo workers from development, quality assurance and rework to testing. This policy-strategy would remove any option for multiple testing stages. The errors found in testing can only be investigated and reworked on site. This policy strategy is pursued when a software project is working under a tight deadline. The main managerial concern is to finish as many development-tasks as possible before the final systems test in order to meet the deadline.



Output (FracOfEffortForSystemTesting...)

Coordinates:

X	Y
0,0	1,00
0,1	0,58
0,2	0,31
0,3	0,14
0,4	0,05
0,5	0,00
0,6	0,00
0,7	0,00
0,8	0,00
0,9	0,00
1,0	0,00

Y:

Points: 11

X-Axis Min: 0,00 Step: 0,10

Y-Axis Min: 0,00 Max: 1,00

Edit What: Graph

Interpolation: Line

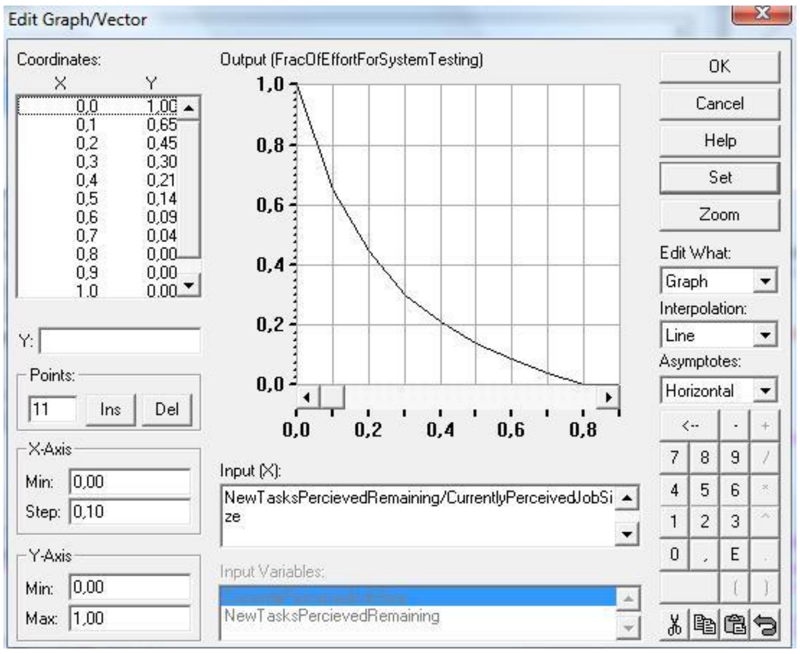
Asymptotes: Horizontal

Input (X): NewTasksPercievedRemaining/CurrentlyPerceivedJobSize

Input Variables: NewTasksPercievedRemaining

Figure 109. Fraction of effort for system testing, Policy 3 strategy2

The policy 3 strategy 2 deployed in this analysis holds the first change in testing-stages as shown in figure 109. For policy 3 strategy 2, the testing effort starts after 50% of all tasks have been completed. The initiation of the testing effort half-way into the software project, in the movement from the design phase to the coding phase of development, provides a testing regime that includes multiple stages. It is able to perform more testing-stages compared to the base model. The table above shows the distribution of manpower to the testing effort. From the Y axis we observe that manpower allocation for testing starts after the 0.5 mark when the project progresses from design to coding. The allocation of manpower increases steadily towards the final systems test.



Output (FracOfEffortForSystemTesting)

Coordinates:

X	Y
0,0	1,00
0,1	0,65
0,2	0,45
0,3	0,30
0,4	0,21
0,5	0,14
0,6	0,09
0,7	0,04
0,8	0,00
0,9	0,00
1,0	0,00

Points: 11

X-Axis Min: 0,00 Step: 0,10

Y-Axis Min: 0,00 Max: 1,00

Edit What: Graph

Interpolation: Line

Asymptotes: Horizontal

Input (X): NewTasksPercievedRemaining/CurrentlyPerceivedJobSize

Input Variables: NewTasksPercievedRemaining

Figure 110. Fraction of effort for system testing, Policy 3 strategy3

The policy 3 strategy 3 presented by the graph in figure 110 is a quality focused policy. In this policy, management tends to regard testing as an important supplement to the software production process. It is a tailored approach for a continuous testing regime that increases as more software is designed and produced. From the graph we see that manpower is allocated to testing as soon as 20% of the tasks have been completed. Under such a strategy, allocation for testing will start much earlier than in the previous two strategies and the base-run.

The completed tasks constitute the first applications ready for testing. The testing crew allocated will start the testing procedures to analyze its applicability. This kind of regime is found in modern medium-range software projects, where each testing step provides a milestone for the project. Management is able to measure when 20% of the job is actually done, since testing starts at this point in time. In the original base-model management had few tools for measuring progress. From Abdel-Hamid’s interviews it was reported that some projects measured progress in production-days. With this testing policy, management is able to measure progress between each testing phase. Compared to strategy 2, the initiation allows even more testing-stages to be performed through the course of software development.

The difference in allocation is visible in the graph, as the curve rises much earlier, and increases smoothly through the project. In the end there is a final systems test as the macro-level defect removal effort, but the software has been tested and corrected several times before this stage is commencing. From the empirical data of Jones and Bonsignour, this scenario is realistic for modern projects (Jones and Bonsignour, 2012). Another important factor in this scenario is the feed-back to the rework and quality assurance system. Testing will reveal through the capture-recapture effort escaped errors that have no tags and bad fixes that have tags. In this way testing, quality assurance and rework supplement each other alongside the production process.

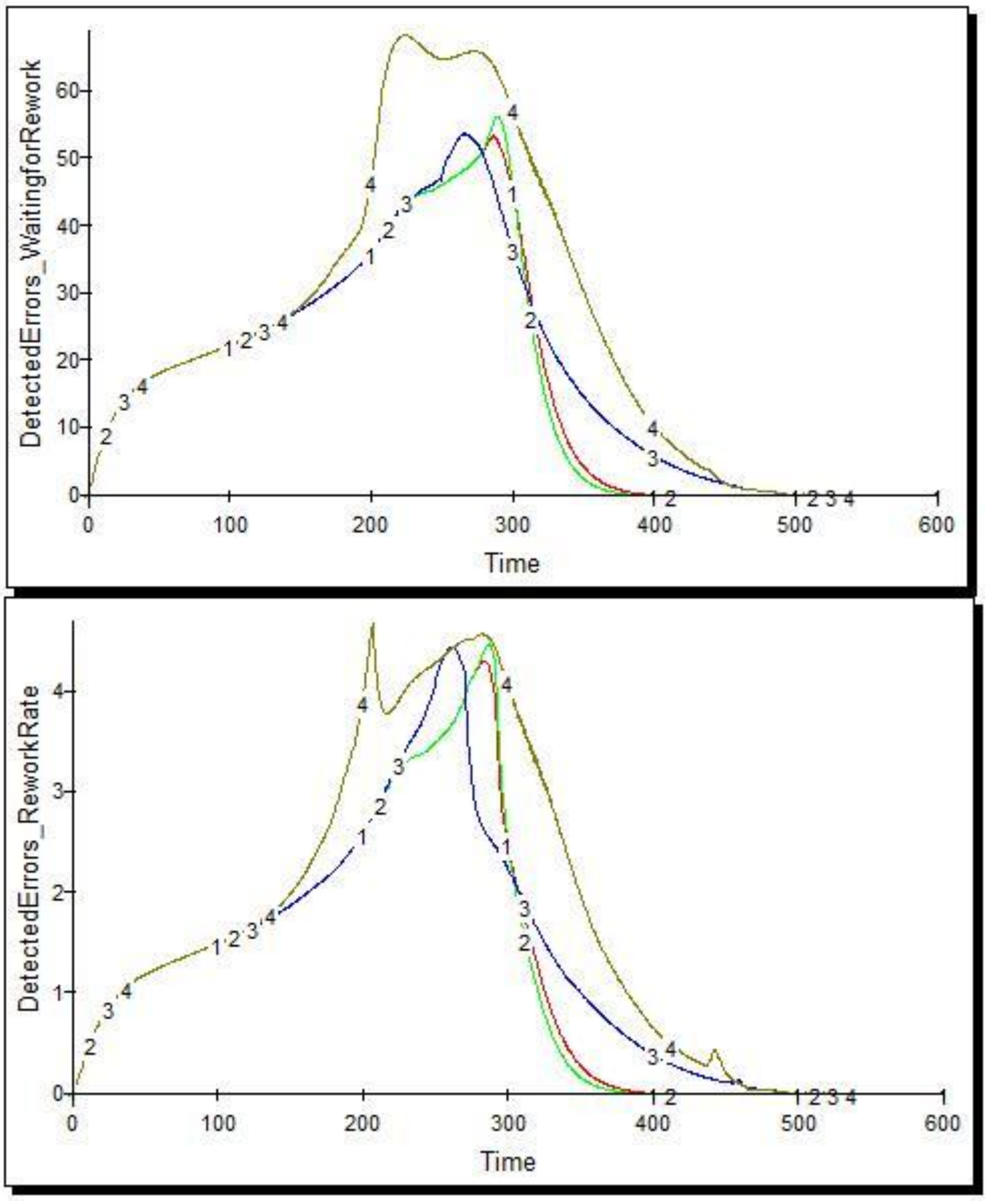


Figure 111. Detected errors waiting for rework and detected errors rework rate, Red line presents the base run, green line presents the strategy1, blue line presents strategy2 and gray line present strategy3

In the original model, 80% of tasks were completed before any personnel was transferred to testing. For strategy 1, the testing design is similar, but manpower is not allocated to testing before 90% of all tasks have been completed. This results in a behavoiural development that follows closely the base run. However, when testing is initiated, a peak of errors are detected that are higher than the base run peak because the rework effort and the quality assurance effort run longer before manpower is reallocated for testing. This is also observable in the rework rate. When testing takes place at a late stage, no feedback from the errors found in testing reaches the rework effort. Hence, the peak increase in the rework rate allows the remaining errors to be removed at a quicker rate.

For strategy 2 the testing-effort starts at the 50% mark of the project. Just as the software development moves from design to coding phase of development, the first testing-effort is initiated. The result is an increase in the detected errors waiting for rework, that surpasses the base-run. The peak leads to an increase in the rework-rate as well, in order to accommodate the level of detected errors waiting for rework. The graph shows that the strategy do improve the detection of errors, indicating that more testing stages are better than a single system testing. However, the increase in detected errors do not exceed the level observed for policy strategy one. Hence, the total amount of errors detected in testing is not increased compared to strategy 1 that initiates testing after 90% of all tasks are finished. Compared to the base-run, the total level of detected errors is not significantly different. The same observation is made in the rework-rate. The inclusion of a testing-effort after 50% of all tasks have been completed yield earlier detection of errors, but not a significant incerase in the total detection of errors. In addition, strategy 3 lacks the capture re-capture effort incorporated in strategy 3.

In strategy 3, the testing effort starts when 20% of the tasks are completely. This is the system tailored for continuous testing and feedback from errors are introduced to either quality assurance or rework. Untagged escaped errors, which are the previous escaped errors from quality assurance, are picked up in testing and sent back to quality assurance. Tagged errors are the result of bad fixes and are subsequently sent to rework. This changes significantly the amount of deteced errors waiting for rework. We observe that the total number of detected errors increases way above the number in the base run due to the factors mentioned. As more errors are detected, more manpower is allocated to the rework effort. This is noticeable in the detected errors rework rate. We see a sharp increase with a peak in the rework rate at an early point.

As the project moves from design to coding we observe a decrease in the detected errors waiting for rework. There two factors explaining this development. The first factor is progress from design to coding. The development staff has accumulated increased knowledge of the sofwtare, and gained valuable experience. In addition coding-tasks are simpler to complete than design tasks. As a result the sofwtare development productivity increases and more tasks are completed. However, due to the incerase in learning and experience the completed tasks contain on average fewer errors. This factor influences the observed decrease in the rate of detected errors waiting for rework. Other important factors are motivational losses, turnovers and information overheads influencing the quality assurance effort. These negative impacts reduce the potential error detection rate for quality assurance, and less errors are detected and sent for rework.

There is an additional effect of the sharp increase in the stock of detected errors and rework rate, that is not observed in the other policy strategies. Due to the peak in detected errors, the rework rate is increased significantly to accommodate the higher level. As a result, the initial peak exhausts the rework-team. The rework-effort is not able to maintain the high rework rate due to exhaustion, motivational losses and turnovers. For a period of time the rework rate drops to a lower level as observed in the graph depicting the rework rate.

However, due to the incerase in error detection, there are still a significant number of errors waiting for rework. This signals to management that an upwards adjustment to the total manpower size is needed to accommodate the high level of errors waiting for rework. The rework rate incerases yet again as a result of these adjustments. Workers cut slack-time and increase their efforts on correcting the remaining errors. The rework rate increases despite previous exhaustion due to an increase in the total workforce level. In addition, the prospect of imminent completion provides positive motivation for the rework staff.

At the end of the project, all development tasks have been completed. Manpower is now transferred from development, quality assurance and rework to testing. There are still tasks that contain errors and needs to be reworked. This effort is completed by the remaining rework personnel, which are incrementally transferred to testing. The result is a behavior pattern in detected errors rework rate and detected errors waiting for rework that decreases in a smoothing fashion towards zero.

Analysis report for completion time, detected errors reworked, man-days expended and benefit-cost ratio

The next step of this analysis is the report for completion time, detected errors reworked and man-days expended. The first table consists of the completion time for each policy strategy and the base-run.

Policy simulation	Completion time
Base run	438 days
Policy 3 strategy 1	437 days
Policy 3 strategy 2	464 days
Policy 3 strategy 3	516.5 days

Figure 112. Project completion time for base run and policy strategies runs

The first observation we make is the scarce difference between the base-model and strategy 1. The choice to halt the allocation of manpower for testing until 90% of all tasks have been completed is not providing a significant benefit in regards to completion time. The strategy leads to a completion date 1 day earlier than for the base-run.

For strategy 2, the earlier testing effort leads to a longer completion time for the project. Since more manpower is moved to the testing effort at an earlier stage, the software development effort is lowered. Fewer tasks are being completed as a result compared to the completion-rate of the base-run. In addition the effort detects more errors, leading to an increase in the total rework level needed to complete all corrections. Yet again, more manpower is diverted from production to defect-removal as a result.

For policy strategy 3 the testing effort is initiated after 20% of all tasks have been completed. This causes a significant increase in the completion time compared to the base model. The continuous testing system redirects even more manpower to the defect removal effort compared to strategy 2. More manpower is allocated for testing, and the rework effort is increased to accommodate the higher level of detected errors waiting for rework. Hence the difference in completion-time between strategies 2 and 3 is 52.3 days, which is also a sizeable difference. Policy strategy 3 enables a higher rate of detected errors but the completion time is significantly longer. Compared to the base-run the difference in completion for strategy 3 is 78,5 days.

The next report is the total defect-removal for each policy strategy, depicted in the table below in figure 113.

Policy simulation	Defect removal
Base run	709,40 errors
Policy 3 strategy 1	702.14 errors
Policy 3 strategy 2	758.65 errors
Policy 3 strategy 3	994.51 errors

Figure 113. Project defect removal for base run and policy strategies runs

From this table the difference in defect removal for each policy strategy suggests a positive effect of the enhanced testing scheme. Detected errors are the benefit-side for a project, and the base-run was able to detect and remove 709.40 errors. Strategy 1 were testing commenced after 90% of all development tasks were completed yielded a lower result of 702-14 errors. For strategy 2 the total amount of errors detected surpass the base run by a clear margin. Earlier testing processes are able to detect more escaped errors and bad fixes in a project. Hence, the total number of reworked errors is higher. This provides better overall software quality for the end product. However, for the significant difference, strategy 3 is displaying the largest improvement by far. The continuous testing system with the capture re-capture process enables a significantly better error-detection rate. The policy strategy is able to detect 994.51 errors which is an increase of 285 detected errors compared to the base run. The second best option in this department is strategy 2 that yields 48 more errors detected.

The third and final table in figure 114 is the total costs for the project measured in man-days. In this table I report the man-day expenditure for each policy strategy and compare them with the base-run.

Policy simulation	Man-day expenditure
Base run	3878,70 Man-days
Policy 3 strategy 1	3865,12 Man-days
Policy 3 strategy 2	4183.28 Man-days
Policy 3 Strategy 3	5460.74 Man-days

Figure 114. Project man-day expenditure for base run and policy strategies runs

The original base-model finished the software project with a man-day expenditure of 3878,70. For strategy 1 the man-day expenditure was slightly lower at 3865,12 man-days expended.

Due to a higher software production rate, and an earlier completion time the costs spent on the project are naturally lower. For strategy 2 where we started allocation in the middle of the project rather end of the project, the increases in error-detection leads to a higher effort in rework. This diverted manpower from production to defect-removal and the completion time was longer than for the base-run. In addition, the allocation of staff to testing at an earlier point in time caused the project to complete at a later date. Combined with a need for manpower adjustments to accommodate a higher level of detected errors waiting for rework, more man-days were expended on the project as a whole. The same pattern occurs for strategy 3, as the testing effort started at earlier stage in the project when 20% of all tasks completed. More effort diverted to defect removal, and more man-days expended as a result.

The difference between policy 3 and the base-run reveals a significant difference due to the continuous testing process. The amount of detected errors was significantly higher, so the rework-effort had to expend more man-days correcting them all. However, even though the costs are increased for both strategy 2 and strategy 3, the change in testing-regimes provided an increase in detected errors as the benefit.

The next part of my analysis is to present the graphs depicting the cumulative detected errors reworked, and the cumulative man-days expended. The tables report the static final results for the benefit and cost element. The graphs on the other hand, depict the dynamic development of the cumulative stocks as the development process progresses. Hence, we can determine at what specific time the impacts of difference in the policy-strategies cause the reported end-result.

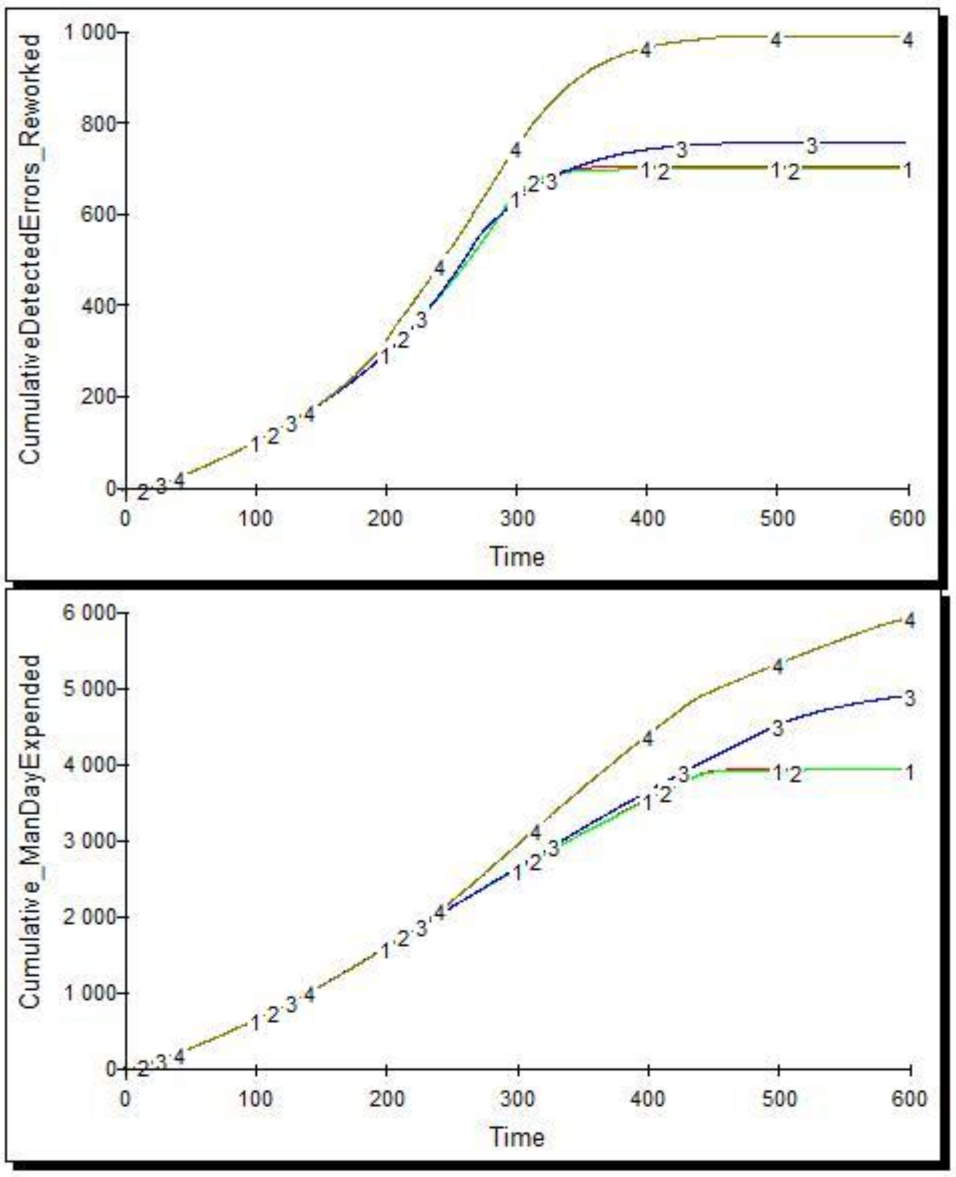


Figure 115. Cumulative detected errors reworked and cumulative man-day expeneded, Red line presents the base run, green line presents the strategy1, blue line presents strategy2 and gray line is strategy3

The results from the two graphs above follow a pattern that is expected based on the reported end-results. The cumulative detected errors reworked follow the same trajectory for my policy strategies compared to the base-model. Between the base-run and strategy 1, there is no significant difference. For strategy 2, there is an increase in the cumulative errors reworked that occurs when the testing effort starts half-way into the project. However, the initial increase is not significant. The change in level occurs at the end of the project, when more manpower has been allocated to testing and the effort is more efficient. The graph for policy strategy 2 levels off at a higher level of detected and corrected errors.

For strategy 3 however, the change in level and trajectory is evident. The testing-effort starts earlier in the project and half-way through the design phase, when 20% of all development tasks are completed. This early detection causes the stock of detected errors waiting for rework to rise. To accommodate this change in detected errors the rework effort is increased. The cumulative rate of reworked errors increases as a result. As progress, experience, productivity and allocation continue to increase the level of detected errors, the trajectory for strategy 3 follows the same pattern as the base-run and the other policy strategies. However, the level is significantly higher.

On the cost side, the second graph in figure 115, we can observe that the graph for the base-run and strategy 1 are similar. The expenditure of man-days is done in the same manner and at the same level. For strategy 2 the man-day expenditure increases slightly as the effort is initiated half-way into the project. However, the increases in costs do not occur before the occurrence of the significant increase in detected errors at the end of the software project. These results are mirroring the total man-day expenditure reported in the last table. For strategy 3, the man-day expenditures increases as soon as the testing effort has been initiated. As explained earlier in this analysis, the higher number of detected errors at an earlier stage causes an increase in the defect-removal effort. As a result more time is spent on the project, and management adjusts the manpower-size in order to correct the high level of detected errors. Hence man-day expenditure follows the same trajectory-pattern as the base-run but at a significantly higher level.

Again as in the previous analyses in the two graphs in figure 115 displayed for cumulative detected errors waiting for rework and for man-day expenditure we observe the characteristic S-shape growth. The behavior pattern initiates in an exponential growth pattern due to increase in activity and progress in software development. As explained in further detail earlier in this chapter, the potential for error detection and software development increases in the early stages of the software project. This increases both the level of cumulative detected errors reworked, and the man-day expenditure. When the project is closer to completion, the goal of zero remaining tasks is closer. This changes the behavior pattern observed in both graphs, as they now display the goal-seeking behavior pattern. This behavior-pattern is consistent between the base-run and the policy strategies.

The final part of this isolated analysis is the benefit-cost ratio. This graph in figure 116 determines the benefit of detecting and correcting more errors versus the cost in man-days. My aim is to provide better quality within a cost-beneficial frame. For a policy strategy to accomplish that, the benefit-cost ratio must be higher than the base run. The graph below depicts the benefit-cost ratio.

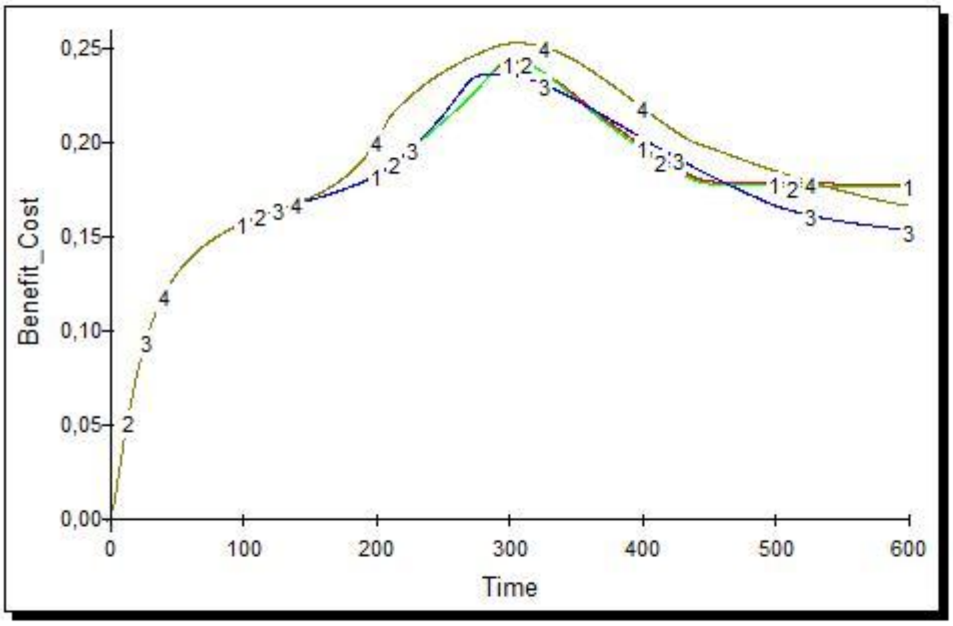


Figure 116. Benefit-cost,
Red line presents the base run, green line presents the strategy1,
blue line presents strategy2 and gray line presents strategy3

From the graph above it is apparent that strategy 3 is the most beneficial policy strategy. This is not surprising taking into consideration the significantly increased level in errors detected. Even though the policy did expend more man-days and ran for a longer time-period, the rate of detected errors outweighs these factors. For strategy 1 and strategy 2, the benefit-cost ratio is insignificant compared to the base-run. For strategy 2 there is an increase at a certain point in time after the testing-effort has been initiated. However, the change is not able to maintain a higher benefit-cost rate as it slips below the level of both strategy 1 and the base-run at the

300 day mark. This isolated analysis provides firm evidence of the benefits of a continuous testing-regime with more testing stages. In addition, the regime is able to combine the strengths of quality assurance, rework, and testing through the capture re-capture procedure. The results are consistent with central literature on the field. This concludes my analysis of policy 3.

8.1.2.3 Policy 4: Policy structure of the client review stage

In the literature on production in general, and in the literature for software project management the importance of the client is evident. From the classic quality thinking presented in the 60’s and 70s by scholars such as Deming and Juran, to the modern quality approaches of Crosby and Peters the focus on the client and his needs are fundamental. The client is the end-user for any product that is being manufactured. Therefore, the client’s needs must be taken into consideration.

In the realm of software management Hjertø argues that the client is a fundamental source for software success. Any piece of software is developed for a specific purpose. A piece of software is a bespoken item, specifically tailored to perform a certain task or function. The client is the final link in the chain that will utilize the software on a daily basis. The client holds intimate knowledge of the needs, and environment that the software will be deployed in. This is a valuable source of information that any software manager should tap into. A piece of software is only successful if it functions in accordance with the client’s need. If the software is well designed but fails to deliver the functions needed, the design cannot save the software from being a failure.

Hence, in modern software projects the inclusion of a test for end-users is common. In the original model of Abdel-Hamid, the client was kept outside the model boundary. In my quest for improved software quality, the client must be taken into consideration. The ability to ensure overall quality of the software is not only depending on the defect-removal level. It is not a success if the program has no errors, but still fails to satisfy the client’s needs.

The client review stage that I incorporate for my model achieves both these goals. The testing-stage is removing the detected significant deviations during the continuous testing effort. The applications that do function with minor deviations are sent for the client test. Hence, the client is given applications that have been thoroughly tested for significant and performance-degrading deviations. The client is then reviewing the software in accordance with the work-environment it will be deployed into. For example will clerking staff at a bank be sent to test a new piece of software for the account-transaction procedure. Their satisfaction with the program or their frustration over user-diminishing problems will detect flaws in the software program.

These flaws are then examined by the testing-staff in order to clarify the underlying deviations causing client dissatisfaction. These errors are then returned to rework. It is impossible to remove all errors from a software project. However, as the spirit of Poka Yoke dictates: “Plan for zero errors”. Adding the client to the testing-effort is an additional tool to find as many errors as possible and rework them. This ensures two levels of software quality. The first level is the actual removal of defects and errors found by the client when testing for specific purposes. The second level lies in client satisfaction. By allowing the client to test the program, and provide reviews I ensure that the client’s needs are met in the final software product. This elevates the chances of success for the software project as a whole.

The inclusion of a client-review stage requires a new policy structure. Policy structures are complete stock and flow systems modeled in order to govern policy-decisions. In system dynamics modeling policies are created as a result of changes in existing variables, or through additional structures that are created to expand the model boundary. When models are subjected to real-world systems, the initial model may be insufficiently scoped. Important stocks and flows that holds an impact on system behavior has been omitted due to model boundaries, or represented as constant in the model. Sterman stress the importance for a system dynamics model to provide a representation of the system that endogenously explains all behavior. (Sterman, 2000) With a limited scope or exogenous variables, the model may fall short of real-world effects on the system it is mimicking. This leads to a loss of realism, and therefore policy-structures are a necessity. In my case we learn from Jones and Bonsignour that client-testing is an important tool in modern software projects. , in recent projects, end-user testing and customer acceptance testing are common. Over 75% of all software projects in the U.S. utilize this form of testing (Jones and Bonsignour, 2012).

In the base model, the client was kept outside the model-boundary. In order to incorporate the client and allow this real-world relationship to be established in my model, a policy structure is an absolute necessity. Hence, I include a complete subsector for the model called “client review”. Now my model holds a complete new structure with stocks, flows and variables that governs this effort. The structure is then attached to the model, and the model boundary has been expanded. This provides my model with realism compared to a real-world software project. The format of this client review is known as the client acceptance test, described by for example Jones and Bonsignour (Jones and Bonsignour, 2012: p.320).

The client-review scheme provides a new policy-structure to my model. The CLD below show how the client-test scheme is related to the testing scheme, and the new loops that it generates.

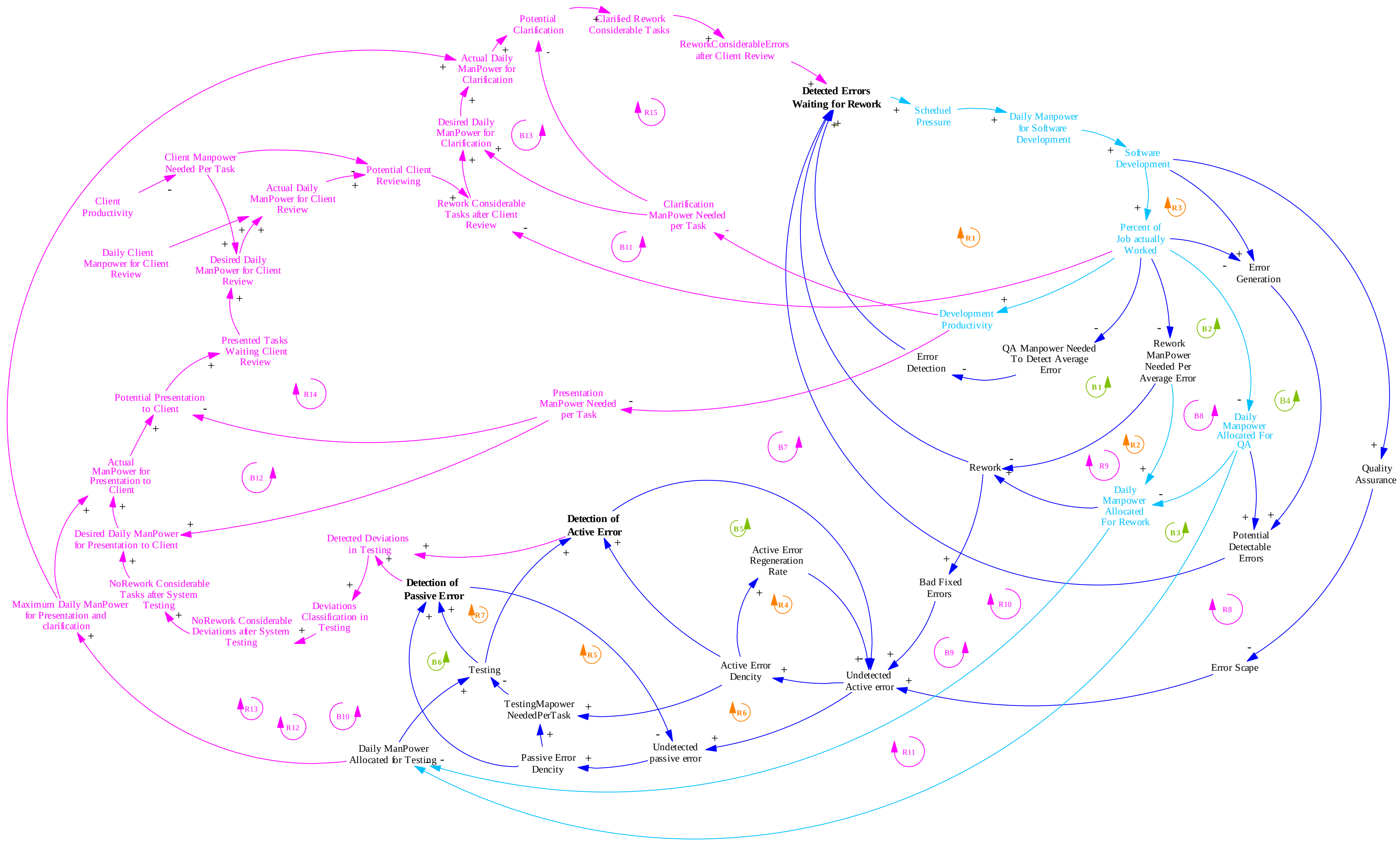
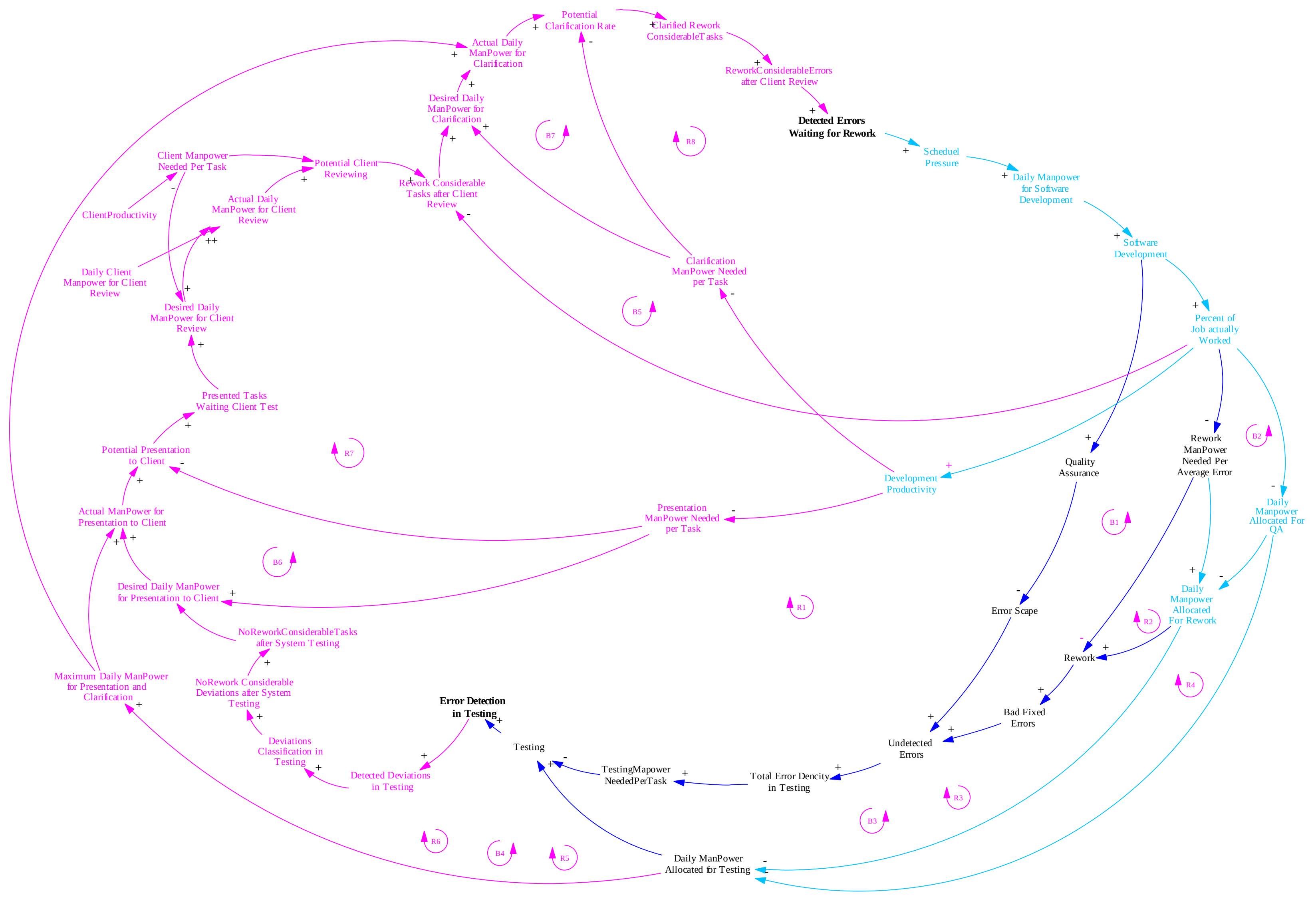

Potential Clarification
Clarified Rework Considerable Tasks
ReworkConsiderableErrors after Client Review
Detected Errors Waiting for Rework
Scheduel Pressure
Daily Manpower for Software Development
Software Development
Percent of Job actually Worked
Error Generation
Development Productivity
Error Detection
QA Manpower Needed To Detect Average Error
Rework ManPower Needed Per Average Error
Daily Manpower Allocated For QA
Rework
Daily Manpower Allocated For Rework
Potential Detectable Errors
Quality Assurance
Bad Fixed Errors
Error Scape
Undetected Active error
Active Error Regeneration Rate
Active Error Dencity
Detection of Active Error
Detection of Passive Error
Testing
TestingMapower NeededPerTask
Passive Error Dencity
Undetected passive error
Daily ManPower Allocated for Testing
Detected Deviations in Testing
Deviations Classification in Testing
NoRework Considerable Deviations after System Testing
NoRework Considerable Tasks after System Testing
Desired Daily ManPower for Presentation to Client
Actual ManPower for Presentation to Client
Potential Presentation to Client
Presented Tasks Waiting Client Review
Presentation ManPower Needed per Task
Desired Daily ManPower for Client Review
Daily Client Manpower for Client Review
Client Productivity
Client Manpower Needed Per Task
Actual Daily ManPower for Client Review
Potential Client Reviewing
Rework Considerable Tasks after Client Review
Desired Daily ManPower for Clarification
Actual Daily ManPower for Clarification
Clarification ManPower Needed per Task
Maximum Daily ManPower for Presentation and clarification
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14
R15
B1
B2
B3
B4
B5
B6
B7
B8
B9
B10
B11
B12
B13

Figure 117. Quality assurance and testing subsectors CLD,

With new loops are provided by client review scheme

In chapter 6, while presenting the enhanced testing-system, I presented a classification process where the significant deviations are classified and identified and sent back to quality assurance or rework. The tasks and applications that are not identified as significant will be sent on from system testing to client-reviewing stage. We observe this relationship in the left-hand side of the CLD, as detection of active and passive errors are fed back to detected errors in waiting for rework. Deviations classified as no rework considerable deviations after classification are gathered into tasks without considerable deviations. These tasks are then presented to the client by the testing-team and client-reviewing commences.

In order to identify the new loops provided by this scheme, and the significant variables the CLD below zooms in on the important variables and structures.



Potential Clarification Rate
Clarified Rework ConsiderableTasks
ReworkConsiderableErrors after Client Review
Detected Errors Waiting for Rework
Scheduel Pressure
Daily Manpower for Software Development
Software Development
Percent of Job actually Worked
Rework ManPower Needed Per Average Error
Daily Manpower Allocated For QA
Daily Manpower Allocated For Rework
Quality Assurance
Development Productivity
Error Scape
Rework
Bad Fixed Errors
Undetected Errors
Total Error Dencity in Testing
TestingMapower NeededPerTask
Testing
Error Detection in Testing
Daily ManPower Allocated for Testing
Detected Deviations in Testing
Deviations Classification in Testing
NoRework Considerable Deviations after System Testing
NoReworkConsiderableTasks after System Testing
Desired Daily ManPower for Presentation to Client
Maximum Daily ManPower for Presentation and Clarification
Actual ManPower for Presentation to Client
Potential Presentation to Client
Presented Tasks Waiting Client Test
Presentation ManPower Needed per Task
Desired Daily ManPower for Client Review
Daily Client Manpower for Client Review
ClientProductivity
Client Manpower Needed Per Task
Actual Daily ManPower for Client Review
Potential Client Reviewing
Rework Considerable Tasks after Client Review
Desired Daily ManPower for Clarification
Actual Daily ManPower for Clarification
Clarification ManPower Needed per Task
B1
B2
B3
B4
B5
B6
B7
R1
R2
R3
R4
R5
R6
R7
R8

Figure 118. New loops are provided by client review scheme

The most significant loops of the policy structure that I have chosen to describe from this CLD are loops R1, B1, R5, B5 and R8.

R1: The first reinforcing loop of this system concerns the software development, quality assurance, escaped errors, errors density and deviations detection in testing, no rework considerable tasks after system testing, rework considerable tasks after client review, rework considerable errors after client review and detected errors waiting for rework.

When software is being developed, the quality assurance effort starts their activity as well. When the quality assurance effort increases, the number of errors escaping detection decreases accordingly. This leads to a decrease in undetected errors and the total error density in testing. The total error density in testing influences the testing manpower needed per task. When the density of errors decreases, the testing manpower needed per task decreases as well, since one task contains strings of codes that have errors. When the effort needed per task decreases, the overall rate of testing increase as a result. This is due to the fact that when the manpower needed per task is lower; the same amount of staff can test and identify more errors than when the effort needed per task is high.

When more testing is being conducted, the deviations detection in testing increases. With more deviations detected in testing, more deviations will be classified during the testing and classification effort. (This) leads to an increase in no rework-considerable deviations after system testing. These deviations are compiled in no rework-considerable tasks that are sent to client review. When the amounts of no rework-considerable tasks are sent to client testing increase, the desired daily manpower for presentation these tasks to client increases. Management allocates a portion of testing staff to prepare client presentation of the application for testing. Increase in daily manpower for client presentation increase the actual manpower for client presentations. When the actual manpower for presentation to client increases, the potential for presentation increases as well. This leads to an increase in presented tasks awaiting client review.

When the tasks have been presented to the clients, the desired daily manpower for client review increases. This leads to an increase in the actual daily manpower for client review and the potential for client reviewing increases as a result. An increase in client reviewing leads to an increase in rework considerable tasks uncovered by the client staff. These tasks leads to an increase in desired daily manpower for clarification, as the testing team must identify and clarify the tasks that have caused the client-staff to consider them unsatisfactory. The increase in desired manpower for clarification subsequently leads to an increase in the actual manpower for clarification. This increases the potential rate of clarification, and tasks clarified as rework-considerable tasks increase subsequently. The identified errors in the rework-considerable tasks increase the number of rework-considerable errors after client review. These errors increase the stock of detected errors waiting for rework. More detected errors waiting for rework increase the schedule pressure, and daily manpower for software development is increased as a result. The loop is closed as the increase in daily manpower for software development increase the rate of software development and subsequently the quality assurance effort.

B1: The first balancing loop of this system concern percent of job actually worked, bad fixes, Errors density, detected deviations in testing, no rework considerable tasks after system testing, rework considerable tasks after client review, rework considerable errors after client review and detected errors waiting for rework.

When software is being developed, the percent of job actually worked increase. The increase signals that the project is moving from the design-stage to the coding stage, and the rework manpower needed per average error decreases. The relationship between these two factors is therefore negative. When the average manpower needed for rework decreases the rate of rework increases. Since more tasks can be reworked per manpower allocated, the same amount of staff is able to process and rework more errors. However when the rework rate increases the chances for bad fixes increases.

Bad fixes occurs when the rework staff erroneously provide a solution to a problem that generates a new problem. This happens for different reasons such as motivational factors or stress. With more bad fixes, more errors are undetected within the tasks, and sent for testing. This leads to an increase in undetected errors. Subsequently an increase in the total error density in testing follows, and requires more testing manpower per task. More manpower needed per average error decreases the rate of testing performed by the testing staff. Less testing leads to less deviation detection in testing. Subsequently, fewer deviations are detected in testing.

When fewer deviations are detected in testing, less deviations will be classified during the testing and classification effort. (This) leads to a decrease in no rework-considerable deviations after system testing. These deviations are compiled in no rework-considerable tasks that are sent to client review. When the amounts of no rework-considerable tasks sent to client testing decrease, the desired daily manpower for presentation of these tasks to client decreases. Management allocates a less portion of testing staff to prepare client presentation of the application for testing. Decrease in daily manpower for client presentation decrease the actual manpower for client presentations. When the actual manpower for presentation to client decreases, the potential for presentation decreases as well. This leads to a decrease in presented tasks awaiting client review.

When the tasks have been presented to the clients decrease, the desired daily manpower for client review decreases as well. This leads to a decrease in the actual daily manpower for client review and the potential for client reviewing decreases as a result. A decrease in client reviewing leads to a decrease in rework considerable tasks uncovered by the client staff. These tasks leads to a decrease in desired daily manpower for clarification, as the testing team must identify and clarify the tasks that have caused the client-staff to consider them unsatisfactory. The decrease in desired manpower for clarification subsequently leads to a decrease in the actual manpower for clarification. This decreases the potential rate of clarification, and tasks clarified as rework-considerable tasks decrease subsequently. The identified errors in the rework-considerable tasks decrease the number of rework-considerable errors after client review and the amounts of detected errors waiting for rework. less detected errors waiting for rework decrease the schedule pressure, and daily manpower for software development is decreased as a result. The balancing loop is closed as the decrease in daily manpower for software development decrease the rate of software development and subsequently the quality assurance effort.

R5: This reinforcing loop of this system concerns percent of job actually worked, the daily manpower allocated for rework, the daily manpower allocated for testing, the allocation of manpower to presentation and clarification process, rework considerable tasks after client review, rework considerable errors after client review and the detected errors waiting for rework.

When the software project is underway and progressing, and the percent of job actually worked increases. This increase leads to less effort needed to rework an average error since project is moving toward the coding phase of development. This causes an adjustment from management, and daily manpower allocated for rework decreases as well. As the rework team belongs to the production effort, the allocation of manpower to rework limits the available staff for testing, as the testing staff is drawn from the production staff.

With an increase in the daily manpower allocated for testing, the maximum daily manpower for presentation and clarification increases as well. With an increase in the maximum daily manpower for presentation and clarification, the actual manpower for presentation to client increase. The rate of potential presentation to client increases accordingly, and the presented tasks awaiting client review follow suit. When tasks are presented, the desired daily manpower for client review increases. This leads to an increase in the actual daily manpower allocated for client review and the rate of potential client reviewing increase as a result. More review increases the number of rework-considerable tasks discovered by the client, and the desired manpower for clarification of these tasks increases as well. Increase in desired manpower, subsequently leads to an increase in the actual daily manpower for clarification. This increases the rate of potential clarification and leads to more clarified rework-considerable tasks. The identified errors in the rework-considerable tasks increase the number of rework-considerable errors after client review. When more rework-considerable errors are clarified after the client test, the detected errors waiting rework increase. This leads to subsequent increases in schedule pressures, and more daily manpower allocated for software development follows as a result. The increase in allocation leads to more software development, and the loop is closed as the adjustment leads to a higher percent of job actually worked.

B5: The fifth balancing loop of the system concerns the percent of job actually worked and the rework-considerable tasks after client review.

As the software project commences, the percent of job actually worked increases. The percent of job actually worked determines the stage of production for the software. The first 50% of the job is the design-phase of development, while the last 50% is the coding phase. When the percentage of job actually worked increases, the project moves from design-phase to coding phase. As the project progresses and the percent of job actually done increases, the number of rework considerable tasks decreases due to the fact of different error-types during the development progress. Design-errors tend to be the source of grave and rework-considerable errors, and when the project has moved to the coding stage, these errors will be in the minority.

The decrease in rework-considerable tasks, leads to a decrease in the desired daily manpower for clarification. With less desire for daily manpower to clarification, the less actual manpower is allocated to the clarification effort. This leads to a drop in the rate of potential clarification and the clarified rework-considerable tasks decrease as a result. This leads to less rework-considerable errors after the client review, and detected errors waiting for rework follow suit. This leads to less schedule pressures, and management adjust the daily manpower allocated for software development downwards. The rate of software development decrease as a result, and the balancing loop is closed when the decrease in rate of software development decrease the percent of job actually worked.

R8: The eighth reinforcing loop of the system links the percent of job actually worked to development productivity, the clarification manpower needed per task and the potential rate for errors clarification.

Increasing the percentage of the job actually worked signals that the project is moving from design to coding phase of development. During the progress, several factors contribute to an increase in development productivity. This increase is due to learning, an increase in workforce experience and easier coding tasks as design tasks require more effort than coding tasks.

This increase in development productivity decrease the clarification manpower needed per task. The reduction in manpower needed per task increase the rate of potential clarification, as the same amount of staff can now work through more tasks and clarify potential errors. The increase in rate of potential clarification leads to an increase in clarified rework-considerable tasks. More tasks rework-considerable leads to more rework-considerable errors after client review and amount of detected errors waiting for rework. The increase in detected errors waiting for rework leads to a subsequent increase in schedule pressure. Schedule pressure leads to an increase in the daily manpower allocated for software development, and more software is being developed as a result. The loop is closed when the increase in software development increase the percent of job actually worked.

The stock and flow chart for this client review structure is presented below and show how members of the testing staff are selected to give a presentation to the client's review staff and then allow them to review the software. The pieces of software provided and tested by system testing are sent from the system testing to client review stage when they are deemed to hold no deviations or minor deviations.

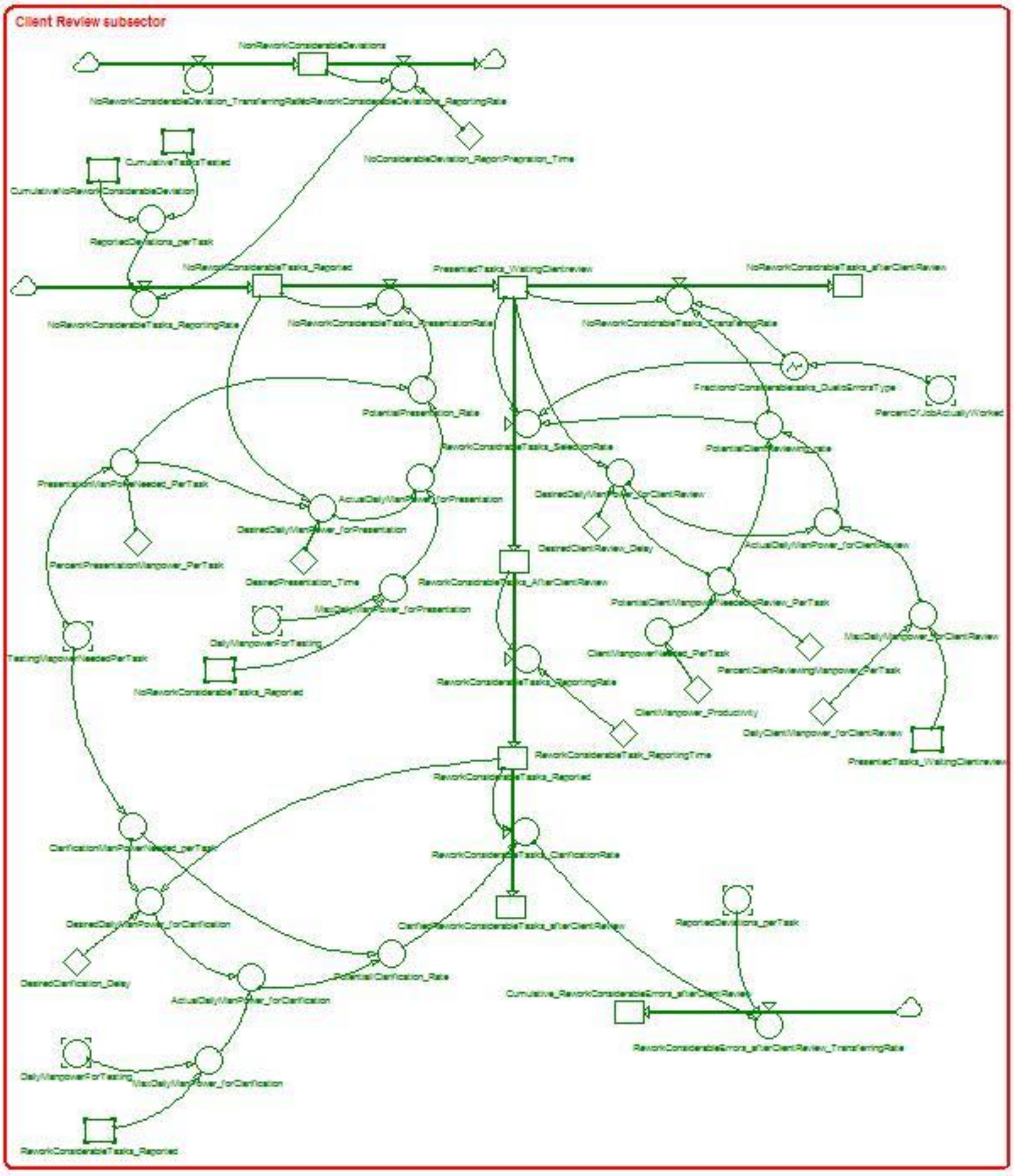

Client Review subsector
NonReworkConsiderableDeviations
NoReworkConsiderableDeviation_TransferringRateNoReworkConsiderableDeviations_ReportingRate
NoConsiderableDeviation_ReportPrepration_Time
CumulativeTasksTested
CumulativeNoReworkConsiderableDeviation
ReportedDeviations_perTask
NoReworkConsiderableTasks_Reported
PresentedTasks_WaitingClientreview
NoReworkConsidrableTasks_afterClientReview
NoReworkConsiderableTasks_ReportingRate
NoReworkConsiderableTasks_PresentationRate
NoReworkConsidrableTasks_TransferingRate
FractionofConsiderabletasks_DuetoErrorsType
PercentOfJobActuallyWorked
PotentialPresentation_Rate
ReworkConsidrableTasks_SelectionRate
PotentialClientReviewing_rate
PresentationManPowerNeeded_PerTask
ActualDailyManPower_forPresentation
DesiredDailyManPower_forClientReview
DesiredDailyManPower_forPresentation
ActualDailyManPower_forClientReview
PercentPresentationManpower_PerTask
DesiredClientReview_Delay
DesiredPresentation_Time
ReworkConsidrableTasks_AfterClientReview
MaxDailyManPower_forPresentation
PotentialClientManpowerNeededtoReview_PerTask
DailyManpowerForTesting
TestingMapowerNeededPerTask
ClientManpowerNeeded_PerTask
MaxDailyManpower_forClientReview
NoReworkConsiderableTasks_Reported
PercentClientReviewingManpower_PerTask
ReworkConsiderableTasks_ReportingRate
ClientManpower_Productivity
DailyClientManpower_forClientReview
ReworkConsiderableTask_ReportingTime
PresentedTasks_WaitingClientreview
ReworkConsiderableTasks_Reported
ClarificationManPowerNeeded_perTask
ReworkConsiderableTasks_ClarificationRate
ReportedDeviations_perTask
DesiredDailyManPower_forClarification
ClarifiedReworkConsiderableTasks_afterClientReview
DesiredClarification_Delay
PotentialClarification_Rate
Cumulative_ReworkConsiderableErrors_afterClientReview
ActualDailyManPower_forClarification
ReworkConsiderableErrors_afterClientReview_TransferringRate
DailyManpowerForTesting
MaxDailyManPower_forClarification
ReworkConsiderableTasks_Reported

Figure 119. Stock and flow diagram for the client review system

The first stage of this new structure is the stock of “No rework considerable deviations” after system testing. When a piece of software has been tested in the testing phase and deemed suitable for client review it is transferred to the stock of “No rework considerable deviations” where the application will be gathered reassembled as a task for the client to review. The transfer rate for tasks between system testing and client review stage rests on the report preparation time. In my model this is assumed to be 3 days.

The inflow of “No rework considerable tasks reporting rate” depends on the outflow of “No rework considerable deviations reporting rate”. Secondly, this rate is determined by the “Reported deviations per task”.

No Rework Considerable Tasks Reporting Rate = No Rework Considerable Deviations Reporting Rate / (Reported Deviations per Task + 0.0001)

The “Reported deviations per task” provides an average density of reported deviations per task that is calculated as a function between the “Cumulative no considerable deviations” after system testing and the “Cumulative tasks tested” after system testing. It is a maximum function and it is calculated as follows:

Reported Deviations per Task = MAX (Cumulative No Rework Considerable Deviation / (Cumulative Tasks Tested + 0.0001), 0)

When an application has been reported and prepared for client review, representatives from the testing staff are sent to meet representatives from the client. They are given a presentation of the finished software application they will review. Before a client team can perform a test and review the usability, they must be briefed on the application function and inner mechanics.

The stock of “No rework considerable tasks reported” present the amounts of reported tasks that should be presented to the clients. The outflow from this stock that is “No rework considerable tasks presentation rate” presents the rate of presenting the tasks to the client. This outflow hinges on the effort needed to present a piece of software to the client. This process is determined by the daily manpower allocated to testing and the testing manpower needed per average task.

This logic follows the same logic as for the testing regime presented in chapter 6 under the topic of enhanced system testing. These two variables are the determinants for the software presentation rate to the client.

This outflow that is "No rework considerable tasks presentation rate", is calculated as follows:

No Rework Considerable Tasks Presentation Rate = (MIN (Potential Presentation Rate, No Rework Considerable Tasks Reported / TIMESTEP))

As is shown in the formula above the outflow depends on the amount of reported no rework considerable tasks that are waiting for presentation. The second element stems from the "Potential presentation rate". There are two determinants for the potential presentation rate. The first is the "Presentation manpower needed per task" and second one is "Actual daily manpower for presentation". These two factors now provide the rate of presentation and pieces of software are presented and given to the client team for reviewing. This "Potential presentation rate" is then linked to the "No rework considerable tasks presentation rate" and functions as the second determinant for this outflow. The variable it is calculated as follows:

Potential Presentation Rate = Actual Daily Manpower for Presentation / Presentation Manpower Needed per Task

To determine the "Presentation manpower needed per task" there are two significant variables. The first is the "Testing manpower needed per task". The formulation of this variable was given in chapter 6 while presenting the earlier enhancement regarding the manpower effort needed per task for the testing process. In short, the manpower needed per task hinges on error density, development productivity and the testing manpower needed per error. The effort needed to present a task is much less than the effort needed to test a task. Therefore, "Presentation manpower needed per task" is determined by the same logic as "Testing manpower needed per task" under the effect of "Percent Presentation manpower per error". In my model this percentage is assumed to be 10%. This variable is calculated as follows:

*Presentation Manpower Needed per Task = Testing Manpower Needed Per Task * Percent Presentation Manpower per Task*

The "Actual Daily Manpower for Presentation" is a minimum function between the desired and the maximum daily manpower for presentation. These variables provide the first half of the calculation for the potential presentation rate, namely the amount of manpower available for presentation. The variable takes the minimum value between the desired and maximum value of daily manpower to control for over-allocation of manpower to presentation stage compared to the desired manpower.

Actual Daily Manpower for Presentation = MIN (Max Daily Manpower for Presentation; Desired Daily Manpower for Presentation)

The “Desired daily manpower for presentation” is determined by the “Presentation manpower needed per task”, “No rework considerable tasks reported” and the “Desired presentation delay”. In my model this delay is assumed to be 3 days. Desired daily manpower for presentation provides the first part of the calculation for “Actual daily manpower for presentation”.

*Desired Daily Manpower for Presentation = (No Rework Considerable Tasks Reported * Presentation Manpower Needed Per Task) / Desired Presentation Delay*

The “Maximum daily manpower for Presentation” rests on the “Daily manpower allocated for testing” and controlled by the stock of “No rework considerable tasks reported”. Daily manpower for testing is determined by management’s allocation of the total effort to testing. The presentation stage is done in tandem with staff from the testing effort. The total amount of staff allocated to testing effort provides the potential workers for the presentation effort as well. An if-function is applied here to control the allocation of manpower to the presentation stage as soon as no rework considerable tasks are prepared for presentation. This variable is calculated as follows:

Maximum daily manpower for presentation = IF (No Rework Considerable Tasks Reported >0; (Daily Manpower for Testing); 0)

During the presentation stage, the client is given a brief introduction to the application and will work more closely with it during the client review stage. When the application has been presented, the software is sent to the next stage of the system. Here the client’s review team will commence the task of reviewing and validating the applications. This is done to make sure they fulfill the client’s needs and expectations. It is also a valuable venture for the project as a whole, as the software is being tested in its end-user environment.

The rate of client reviewing hinges on the client team’s allocation and ability to work through the applications provided. I will here only present the important determinants for the rate of client reviewing. The managerial policies behind allocation and effort from the client side are exogenous in this model. The dynamic behind the client’s decision for allocation is outside the model boundary.

When the tasks have been sent to client review stage, the assigned client staff begins the process of further reviewing the software. Applications tested by the client team will be classified as either suitable, containing non considerable deviations and non-suitable with considerable deviations. Here the end-user can determine if flaws or the minor shortcomings are considerable enough to degrade the software’s implementation or user-experience.

Suitable tasks hold no considerable deviations and they do perform the assigned function. Sometimes they do perform with minor shortcomings or flaws but they meet the client’s needs and orders. So the client considers these tasks as suitable tasks that need no further rework.

Non-suitable tasks according to the client team consideration are the tasks that are pieces of software that do not function according to the design or they do not meet the client’s needs and orders. Therefore, the client considers these tasks as non-suitable tasks that need further rework.

The factors and processes above are presented as the client review stage in the model. The outflows for the stock of “Presented Tasks Waiting Client Review” are “No rework considerable tasks” and “Rework considerable tasks”. Both these outflows hinge on the daily client manpower for client review, client productivity, client manpower needed per task and the percent of job is actually done.

The first outflow that is “Rework considerable tasks selection rate” that is calculated as follows:

*Rework Considerable Tasks Selection Rate = (MIN (Potential Client reviewing rate, Presented Tasks Waiting Client Review / TIMESTEP)) * Fraction of Considerable tasks Due to Errors Type*

As is shown in the formula above the outflow depends on the amount of presented tasks waiting for client review. The second element stems from the “Potential client reviewing rate” that exists in the client review stage. There are two determinants for the potential client reviewing rate. The first is the “Potential client manpower needed to review per task” and second one is “Actual daily manpower for client review”. This “Potential client reviewing rate” is then linked to the “Rework considerable tasks selection rate” and functions as the second determinant for this outflow. The variable it is calculated as follows:

Potential client reviewing Rate = Actual Daily Manpower for client review / Potential client manpower needed to review per task

To determine the “Potential client manpower needed to review per task” there are two significant variables. The first is the “Client manpower needed per task”. The client manpower needed per task hinges on client manpower productivity. Client manpower productivity is assumed to be an exogenous variable in my model that is set to 1 task per man-day. The effort needed to review a task is much less than the effort needed to develop a task. Therefore, “Client manpower needed per task” is under the effect of “Percent client reviewing manpower per task”. In my model this percentage is assumed to be 10%. This variable is calculated as follows:

*Potential Client Manpower Needed to review per Task = Client Manpower Needed Per Task * Percent Client Reviewing Manpower per Task*

The “Actual Daily Manpower for Client review” is a minimum function between the desired and the maximum daily manpower for client review. These variables provide the first half of the calculation for the potential client reviewing rate, namely the amount of manpower available for client reviewing. The variable takes the minimum value between the desired and maximum value of daily manpower to control for over-allocation of manpower to client reviewing stage compared to the desired manpower.

Actual Daily Manpower for Client review = MIN (Desired Daily Manpower for Client review, Max Daily Manpower for Client Review)

The “Desired daily manpower for client Review” is determined by the “Potential client manpower needed to review per task”, “Presented tasks waiting client review” and the “Desired client review delay”. In my model this delay is assumed to be 3days. Desired daily manpower for client Review provides the first part of the calculation for "Actual daily manpower for client Review".

*Desired Daily Manpower for Client review = (Presented tasks waiting client review * Potential client manpower needed to review per task) / Desired Client review Delay)*

The "Maximum daily manpower for client review" rests on the "Daily client manpower for client review" that is assumed to be an exogenous variable in my model and is set to 2 man-days per day. The "Maximum daily manpower for client review" is controlled by the stock of "Presented tasks waiting client review". An if-function is applied here to control the allocation of manpower to the client review stage as soon as presented tasks are prepared for client review. This variable is calculated as follows:

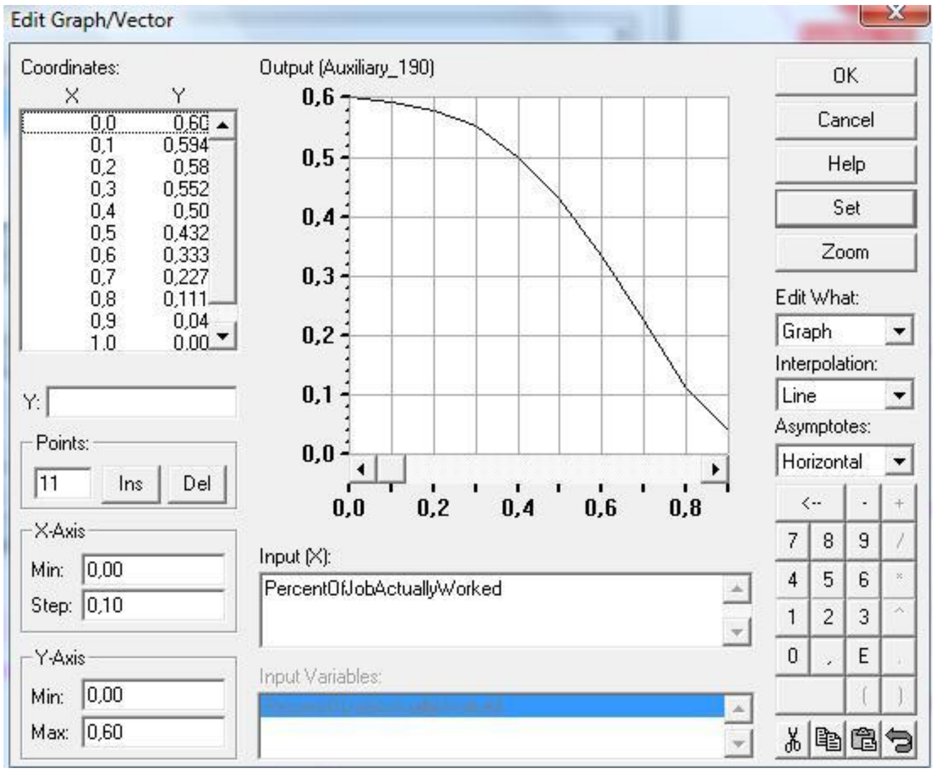
Maximum daily manpower for client review = IF (Presented Tasks Waiting Client Test >0; (Daily Client Manpower for Client Review); 0)

The fourth and final determinant for the out flow of "Rework Considerable tasks selection rate" is the percent of job actually worked through the "Fraction of considerable tasks due to errors type". From previous explanations in chapter 6, I explained how the percentage of job actually done determines the stage of production for the software. The progress represents the movement of the production phase from design to coding. The first 50% of the job is the design-phase of development, while the last 50% is the coding phase.

Coding tasks that contain coding errors are by nature easier to detect and fix than design tasks that contain design errors in the previous stages of quality assurance, rework and system testing. Therefore, when the project progresses and the percentage of job increases, the amount of rework considerable tasks selected by the client decreases due to the effect of different types of the tasks and errors along the development phase. Hence, this variable is the final determinant to the flow between "Presented tasks waiting client test" and "Rework considerable tasks after client review" sent to the identification stage.

This is an important factor to take into consideration, as the design phase provides different challenges to client classification than the coding stage. When a project is being designed, the functions and blueprint of the software is constructed according to the preliminary plan. In this effort, the framework for the project is still loosely based on the initial wishes of the client. The client has a vision for the end-product, and in the design effort these visions are put into practice through the development of a software blueprint. The initial tasks presented for the client are therefore more technical and demanding to rework and classify for a client reviewing team. Since the design is still open, the client also holds the opportunity to add new features or tasks that the software should be able to cover. The effort is therefore initially harder, and this is identified in the fraction of considerable tasks in client reviewing (Jones, 2007).

This multiplier follows nearly the same logic and pattern as the percentage of active errors compared to amount of job is worked in the base model. (Abdel-Hamid, 1991: 112) During the design phase the multiplier holds a factor of 0.6, as the tasks sent for client review are design tasks and contain design errors that are harder to detect and fix in the previous stages of quality assurance, rework and system testing. When the project reaches the coding-phase, the amounts of tasks are increasingly coding tasks and the need for rework consideration tasks is reduced steadily until it reaches 0 at the end of production. When the codes and applications have been finished, the software is more concrete. When the software is concrete, the client review team will have an easier task of testing the software and classify it. It is either suitable to their needs, or they will have specific requests for changes that are sent to rework. This multiplier is depicted below:


Edit Graph/Vector

Coordinates:

X	Y
0,0	0,60
0,1	0,594
0,2	0,58
0,3	0,552
0,4	0,50
0,5	0,432
0,6	0,333
0,7	0,227
0,8	0,111
0,9	0,04
1,0	0,00

Output (Auxiliary_190)

Points: 11

X-Axis Min: 0,00 Step: 0,10

Y-Axis Min: 0,00 Max: 0,60

Input (X): PercentOfJobActuallyWorked

Edit What: Graph

Interpolation: Line

Asymptotes: Horizontal

Figure 120. Fraction of considerable tasks due to tasks type

The second outflow for the stock of “Presented Tasks Waiting Client Review” “No rework considerable tasks transferring rate”. This outflow also hinges on the daily client manpower for client review, client productivity, client manpower needed per task and the percent of job is actually done. This outflow is calculated as follows:

*No Rework Considerable Tasks Transferring Rate = (MIN (Potential Client reviewing rate, Presented Tasks Waiting Client Review / TIMESTEP)) * (1-Fraction of Considerable tasks Due to Errors Type)*

As is shown in the formula above, the outflow depends on the amount of presented tasks waiting client review. The second element stems from the “Potential client reviewing rate” that exists in the client review stage. The third determinant for the out flow is the percent of job actually worked through the “Fraction of considerable tasks due to errors type”. During the presentation of the first outflow, these three determinations where explained in details.

When a piece of software or an application has been reviewed by the clients and classified as unsuitable the testing staff must clarify the underlying errors that gave the clients the problematic experience with the application. The client staff is not technical, and will only provide the testing-staff with their verbal experience with the software. The rework considerable tasks are reported to the testing staff. The “Rework Considerable Task Reporting Time” is set to 2 days in my model. The next step in the process is for the testing staff to clarify the errors in the task.

The stock of “Rework considerable tasks reported” presents the amounts of reported tasks that should be clarified by the testing staff. The outflow from this stock is “Rework considerable tasks clarification rate” and presents the rate of error clarifying the tasks. This outflow hinges on the effort needed to clarify the rework considerable tasks. This process hinges on the daily manpower allocated to testing and the testing manpower needed per task.

This logic follows the same logic as for the presentation stage. These two variables are the determinants for the errors clarification of the rework considerable tasks.

This outflow is “Rework considerable tasks clarification rate” and it is calculated as follows:

Rework Considerable Tasks clarification Rate = (MIN (Potential clarification Rate, Rework Considerable Tasks Reported / TIMESTEP))

As is shown in the formula above the outflow depends on the amount of reported rework considerable tasks that are waiting for clarification. The second element stems from the “Potential clarification rate”. There are two determinants for the potential clarification rate. The first is the “Clarification manpower needed per task” and second one is “Actual daily manpower for clarification”. These two factors now provide the rate of clarification and errors inside the rework considerable tasks are clarified and given to the rework team afterwards to get rework. This “Potential clarification rate” is then linked to the “Rework considerable tasks clarification rate” and functions as the second determinant for this outflow. The variable it is calculated as follows:

Potential Clarification Rate = Actual Daily Manpower for Clarification / Clarification Manpower Needed per Task

The “Clarification manpower needed per task” is determined by “Testing manpower needed per task”. The formulation of this variable was given in chapter 6 while presenting the earlier enhancement regarding the manpower effort needed per task for the testing process. In short, the manpower needed per task hinges on error density, development productivity and the testing manpower needed per error. The effort needed to present a task is much less than the effort needed to test a task. Therefore, “Clarification manpower needed per task” is determined by the same logic as “Testing manpower needed per task” as clarification activities are a kind of testing activities that is commenced after client review. The same amounts of efforts as system testing are needed here to clarify the errors inside the tasks as well. This variable is calculated as follows:

Clarification Manpower Needed per Task = Testing Manpower Needed Per Task

The “Actual Daily Manpower for Clarification” is a minimum function between the desired and the maximum daily manpower for clarification. These variables provide the first half of the calculation for the potential clarification rate, namely the amount of manpower available for clarification. The variable takes the minimum value between the desired and maximum value of daily manpower to control for over-allocation of manpower to clarification stage compared to the desired manpower.

Actual Daily Manpower for Clarification = MIN (Desired Daily Manpower for Clarification, Max Daily Manpower for Clarification)

The “Desired daily manpower for clarification” is determined by the “Clarification manpower needed per task”, “Rework considerable tasks reported” and the “Desired Clarification Delay”. In my model this delay is assumed to be 6 days. Desired daily manpower for clarification provides the first part of the calculation for “Actual daily manpower for clarification”.

*Desired Daily Manpower for Clarification = (Rework Considerable Tasks Reported * Clarification Manpower Needed Per Task) / Desired Clarification Delay*

The “Maximum daily manpower for Clarification” rests on the “Daily manpower allocated for testing” and controlled by the stock of “Rework considerable tasks reported”. Daily manpower for testing is determined by management’s allocation of the total effort to testing. The clarification stage is done in tandem with staff from the testing effort. The total amount of staff allocated to testing effort provides the potential staff for the clarification effort as well. An if-function is applied here to control the allocation of manpower to the clarification stage as soon as rework considerable tasks are prepared for clarification. This variable is calculated as follows:

Maximum Daily Manpower for Clarification = IF (Rework Considerable Tasks Reported >0; (Daily Manpower for Testing); 0)

When the client has reviewed the applications, testing staff clarified the errors inside the rework considerable tasks that are considered by client team. With manpower allocated to this clarification effort, the outflow of rework considerable tasks will commence as the testing staff clarifies the underlying errors inside these tasks experienced by the client-review team. From here the application will be transformed into a set of reported errors that will be sent back to rework for fixing.

In the last structure of this system, we see how the outflow of "Rework Considerable Errors after Client Review Transferring Rate" is transferred back to rework. It reappears as an inflow to the detected errors waiting for rework in the quality assurance and rework subsector sector. The process of dismantling the application into raw code with errors is modeled in the bottom right corner of this subsystem. "Rework considerable tasks clarification rate" together with "Reported deviations per task" provide the final outflow of client subsystem that is fed back to rework process. It is calculated as a function:

*Rework Considerable Errors after Client Review Transferring Rate = Rework Considerable Tasks Clarification Rate * Reported Deviations per Task*

<u>Policy analysis for the client review stage</u>

The final isolated analysis for this chapter is the inclusion of the client review stage. The effort has been described thoroughly in the previous part of this chapter, and I am now able to analyze it. Unlike the previous isolated analyses in this chapter, the client review system can only be deployed for a policy-strategy containing a continuous testing system. The effort depends on the early allocation of manpower to testing in order for this client effort to hold any effect. If it was deployed in the original model, the client would not be able to provide his opinion during production, as the base-model only contain a single system testing. Client demands would be incorporated post-production, outside the framework of this model. My aim to increase end-product quality would not be accommodated by this approach, as the product would have been completed by the time the client reviewed it. Post-production maintenance is a symptom of the software crisis my effort is trying to address.

Due to these facts, the analyses of policy four will be conducted while policy 3 strategy 3 is enabled. This strategy contains the continuous testing effort that is activated earlier in the project, in the middle of design phase of development, when 20% of all tasks have been accomplished. During production pieces of software are integrated into working applications. After testing some applications will contain no errors, or smaller deviations. These pieces of software are then transferred to the client review, in what is known as a client acceptance test. The concept of a client review is established by Jones and Bonsignour, and the effort is able to detect and remove 30% additional errors from a software product. This is the client review process that will be deployed for policy 4.

There is a second important factor regarding the client review stage. The manpower that is allocated to the effort stems from two different sources. One half of the manpower is allocated through management to the enhanced testing effort. The second half is the client manpower allocated from the client's staff. The allocation of client manpower and the costs caused by this effort is covered by the client, and not by software project management. The amount of effort dedicated is decided by the client himself. There is no literature on the field that suggests a system where software management is permitted to designate client manpower for a client acceptance test.

The base run of the model follows the original formula reported in the previous isolated policies. "The fraction of effort for system testing" allocates manpower to the testing effort after 80% of all tasks have been completed. The "desired rework delay" is set to 15 days, while the "planned fraction of manpower to quality assurance" is set to 15% of the total daily manpower after training.

For policy 4 the allocation of manpower to testing is activated earlier in the project, in the middle of design phase of development, when 20% of all tasks have been completed. In addition, manpower from the testing effort is sent for the client review effort as described in the presentation of the client review. In the graphs presented in figure 121, the first red line tagged with number "1" is the original base-run. The green line tagged with number "2" is the policy 4 client-review simulation. The first two graphs presented below, are the two variables depicted in my reference mode. The first graph is the development in detected errors waiting for rework, while the second graph depicts the rework rate.

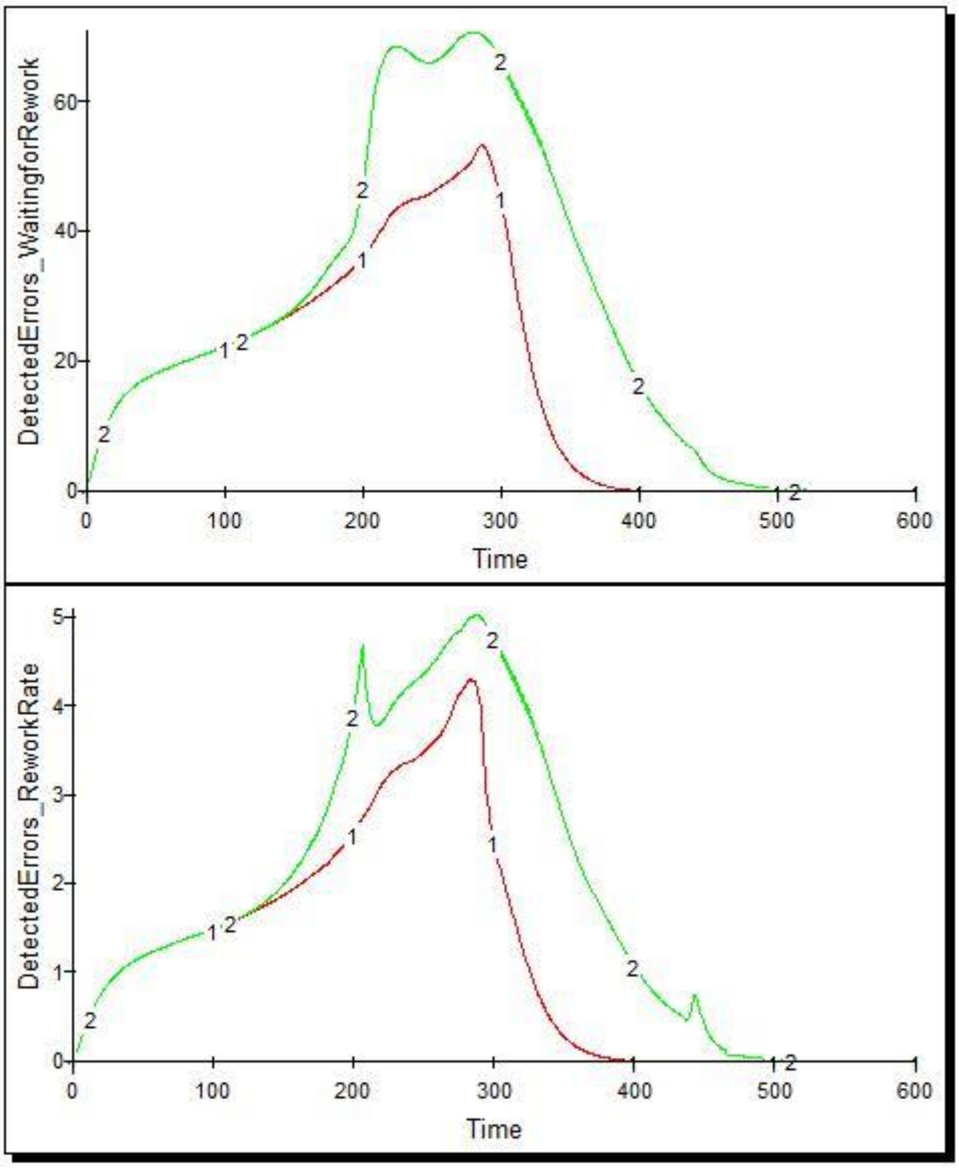


Figure 121. Detected errors waiting for rework and detected errors rework rate, Red line presents the base run and green line presents the policy run

From the graph above in figure 121, we observe that initially the development in both detected errors waiting for rework and detected errors rework rate, are similar for policy 4 and the base run. However, as the testing effort is initialized when 20% of all tasks are completed the detected errors and rework rate increases beyond the base-run. In the middle of the design-phase, the testing effort initiates. Through the testing of applications, additional errors are found that are caused by either previous bad fixes or previous escaped errors from the quality assurance effort. The result is a sharp increase in the amount of detected errors waiting for rework. The increase in the detected errors signals for management that there is a need for a strengthened rework effort. The rework teams cut slack-time, and the rework rate increases as a response to the new amount of detected errors waiting for rework. This level is significantly higher compared to the base-run, and therefore management re-adjusts the manpower-size in order to allocate more manpower for the effort.

The rework rate increases drastically for a period of time. The effect of cutting slack-time and schedule pressures propels the effort forwards. However, this level of rework-rate is impossible to maintain for a long period of time. Exhaustion, motivational losses and turnovers affect the rework rate. This is visible in the rework-rate depicted above. The initial increase peaks off after a period of time due to these factors mentioned above. The rework rate lowers to a level that is tolerable for the exhausted rework team.

At this point in time we observe that compared with policy 3, there is a significant second peak in the detected errors waiting for rework. In policy 3 strategy 3, the peak of the detected errors waiting for rework is slight and less at this juncture. In the graph above however, the amount of detected errors waiting for rework increases into a second significant peak. The reason for this additional peak is the client review effort. The client has been given several pieces of applications to test and report on. Deviations that are reported from the client’s reviewing-team is analyzed and clarified by the testing staff designated to the client review. These deviations that diminished the usability of the program for the client are now returned to rework.

It is important to point out a factor here regarding the client-review. The effort is not run as a single client test, but a continuous process of application testing. The reason for the late entry of testing-errors detected lies in delay. Tasks are tested by system testing first, then when a piece of software is ready for client test, it is first reported and presented by the testing staff to the client review staff. Then the client tests the program and reports back in their review. Feedback on eventual problems of usability for the client is then analyzed by the designated testing-staff. When these deviations have been identified, they are sent back to the rework effort. This process takes time, and due to these information and process delays, the effect of the client-review enters to the rework by a delay, after the first feedbacks from system testing entered to the rework.

The effect of this client review is visible in both the detected errors waiting for rework, and the detected errors rework rate. Since more errors are detected, the stock of detected errors waiting for rework increases. This provides more tasks still remaining to be completed before the project is finished. Again management must re-adjust the manpower-allocation to the rework effort in order to complete the project on time. By comparing the rework-rate between policy 3 strategy 3 and policy 4, the rework rate increases again sharply in policy 4 at a stronger rate. This is due to the client review's ability to detect more errors, and the defect removal effort must be increased to accommodate this higher level of detected errors.

Compared to the base-run, the level of detected errors and rework rate are significantly higher. As a result there are more errors waiting to be reworked after the software tasks have been completed. Hence, near to the end of the project, as staff is moved from rework to the final system testing the detected errors waiting for rework and the rework rate slowly declines towards zero. Compared to the base-run this effort takes longer time, due to the significantly higher level of detected errors from the testing and the client review process. As a result the project finishes at a later stage than the base run.

<u>Analysis report for completion time, detected errors reworked, man-days expended and benefit-cost ratio</u>

The first table in figure 122 that I present in this analysis, is the reported completion time. One of the important aspects for a software project is the total software production time. The original base-run was able to complete all tasks in 438 days. In this isolated analysis we combined policy 3 strategy 3 and policy 4. The table below reports the completion date.

Policy simulation	Completion time
Base run	438 days
Policy 4	502.5 days

Figure 122. Project completion time for base run and policy run

The results reported in the table above are consistent with the development observed in the graphs for detected errors waiting for rework, and the detected errors rework rate. The continuous testing effort and the inclusion of a client-review stage, increases the total time spent on the defect removal effort. The project is completed after 502.5 days compared to the original 438 days in the base run. By allocating more manpower earlier to the testing effort, the manpower available for software development is lower. Hence, the potential software development rate is lower for policy 4 than for the base-run. This fact in tandem with the increased amount of detected errors, causes the project to run for a longer time period for policy 4.

There is another feature by this completion time that is very interesting. When we compare the completion time for policy 3 strategy 3 with policy 4 it is evident that policy 4 provides an earlier completion time. The amount of errors detected and reworked is higher for policy 4, but it is able to complete the project earlier than policy 3 strategy 3 by 14 days. There are two factors that are related to one-another that explains this phenomenon. The client review is a continuous effort in the same way as the enhanced testing effort. This provides management with better information to the perceived time still needed to complete the project. The available information on errors provided by the enhanced testing effort and the client review allows management's perception of rework-rate needed to be more accurate.

Secondly the addition of errors found by the client review towards the end of the project, lead management to adjust the rework-staff upwards to accommodate the increase in errors. As a result the total amount of manpower allocated to rework at the end of the project is higher than for policy 3 strategy 3. When manpower is transferred to testing, the continuous system has allocated workers to this effort previously after 20% of all tasks where completed. Hence, the sudden need for manpower to the testing effort at the end of the project is lower than in the base-run. When manpower is transferred between development, quality assurance and rework to testing, near at the end of the project, the perceived level of rework still needed change the distribution of manpower-transfer. More rework-personnel is available to finish the last correction tasks, and the project finishes at an earlier point in time compared to policy 3 strategy 3 that incorporates the same continuous effort but without the client review. This factor establishes the beneficial information-side of the client review effort. Management is able to allocate more accurately to the rework effort when more information on the actual need is available.

The second table in figure 123 that is provided in this analysis, concerns the total amount of errors that have been detected and corrected through the software production cycle. For this policy to provide better software quality, the total number of errors removed from the software must be significant compared to the original model. The table below contains the total amount of errors detected and removed. This factor constitutes the benefit side of the project.

Policy simulation	Defect removal
Base run	709,40 errors
Policy 4	1098,38 errors

Figure 123. Project defect removal for base run and policy run

As indicated by the development of detected errors waiting for rework, policy 4 is able to detect more errors than the base run. This causes an increase in the rework-effort, and more errors are detected and removed from the software product. The difference between the original 709.40 errors and policy 4's amount of 1098.38 errors is significant. Compared to the reported errors from policy 3 strategy 3, the addition of the client-review ensures better end-product quality as the total amount is higher. Policy 3 strategy 3 alone without the client review allowed 994.51 errors to be removed. The amount of additional errors detected and removed through the client-review stage is consistent with the empirical data provided by Jones and Bonsignour regarding the effect of defect-removal for client acceptance testing (Jones and Bonsignour, 2012: p.320). Policy 4 is able to detect an additional 103,87 errors compared with policy 3 strategy 3. The client review is a significant entry to the ability of achieving increased software product quality.

The next element that is analyzed here concerns the man-days spent on the project. The project's cost is measured in the total amount of man-days expended on the project as a whole. For any policy to be cost-efficient, the effort of defect removal must hold a higher benefit than the additional cost of redistribution of manpower from development to defect removal. The table below in figure 124, contain the total man-days expended in policy 4 and the base-run.

Policy simulation	Man-day expenditure
Base run	3878,7 Man-days
Policy 4	5429,58 Man-days

Figure 124. Project man-day expenditure for base run and policy

From the development in the graphs depicting the detected errors waiting for rework and the rework rate these results are as expected. In the analysis we observed that the inclusion of a continuous testing regime and a client review stage significantly altered the level of detected errors waiting for rework. As a response, management re-adjusted the allocation of manpower and staff size for the defect-removal effort in order to maintain an efficient defect-removal effort. The extra manpower, and the longer project development time reflects the man-days expended reported in the table above. The base-run expended a total of 3878,70 man-days during the software production cycle. Policy 4 ran for a longer time-period and expended a total of 5429.58 man-days. There is a significant difference in costs between the original base-model and policy 4.

The next part of this analysis addresses the development of benefit and costs through the software project. The tables above provide the static report of the final result. The next two graphs depict the development of errors removed through the “cumulative detected errors reworked” and the expenditure of man-days through the “Cumulative Man-days expended”. Hence, these graphs are able to describe the development of error-removal and expenditure of man-days dynamically through the project’s life-cycle.

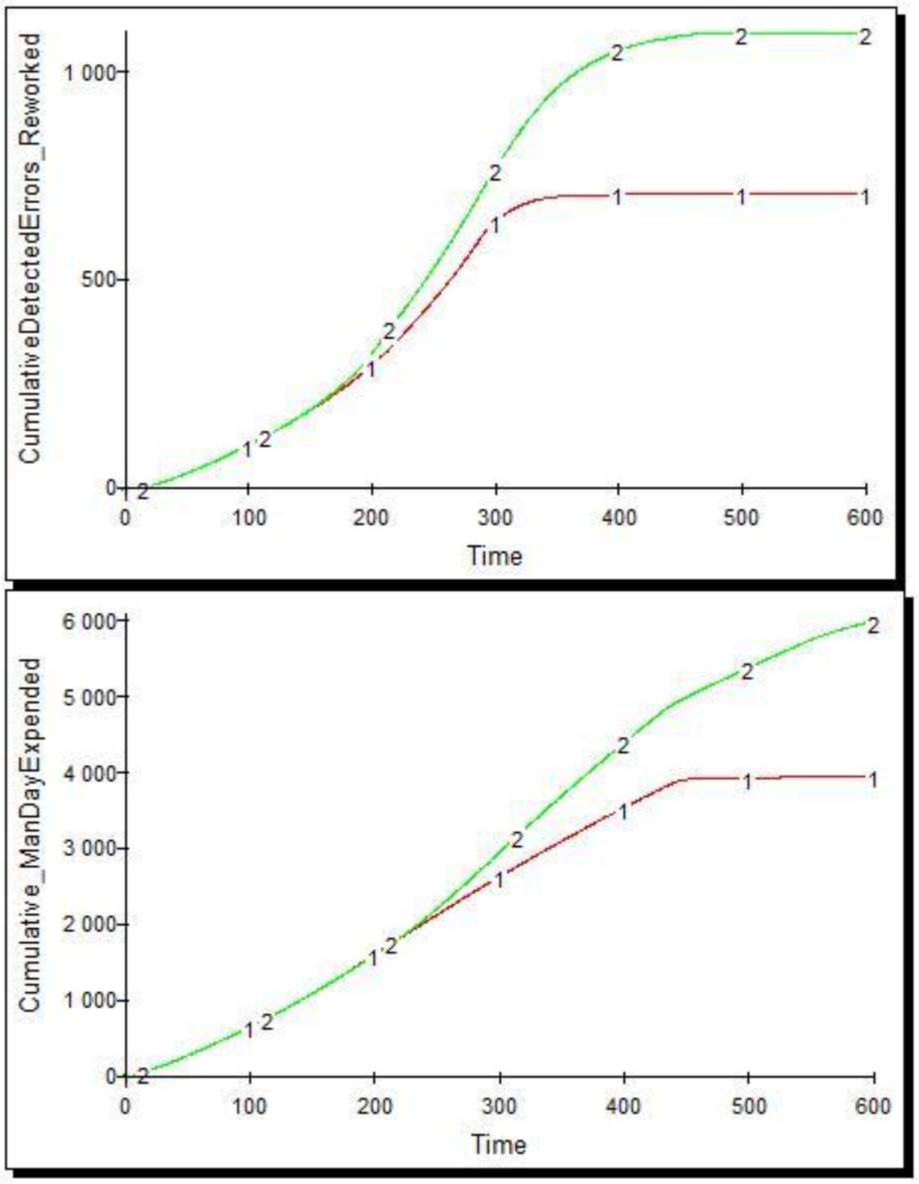


Figure 125. Cumulative detected errors reworked and cumulative man-day expended, Red line presents the base run and green line presents the policy run

As shown in figure 125, compared to the base-run, the cumulative detected errors reworked increases sharply at the introduction of the testing-effort. When 20% of all tasks where completed, testing commenced and more errors were detected. This caused a significant increase in the rework-rate, and the result is visible in the first graph above. In addition to the first increase, there is a second increase occurring towards the end of the project. The client review detects an additional amount of errors, as reported in the table of total amount of errors reworked. The total level of cumulative errors reworked levels off at a significantly higher rate compared to the base-model.

As shown in figure 125, for the expenditure of man-days the same pattern emerges as for the cumulative detected errors reworked. The initial man-day expenditure between base-run and policy 4 are equal until the testing effort is initiated. The testing effort detects an increased amount of errors, causing a re-adjustment towards the rework effort. Management hires more people to accommodate the new level of detected errors waiting for rework, and more man-days are expended. The project continues to run for a longer period of time, causing additional costs. Towards the end of the project the man-day expenditure starts to level off, but still increases slowly. This is the effect of the client-review stage. Since the cost of the client-review is not covered by the project, the increase in expended man-days is not as profound at this stage. However, the increase is significant compared to the base-run. This result is reflected in the table shown in figure 124.

As in the previous analyses in this chapter, the two graphs above depict the characteristic behavioral pattern for development in the cumulative detected errors reworked, and the cumulative man-day expenditure. The initial development follows that of exponential growth for both base-run and the policy strategy. However, the difference in allocation changes the level for the policy strategy compared to the base-run. As the project progresses the end, it reaches its goal of zero tasks remaining. At this point the increase in man-days expended and detected errors waiting for rework levels off towards the goal. This provides the S-shaped pattern discussed earlier in this chapter.

There is no question that the testing-regime and client review is able to detect more errors. At the same time, the cost in man-days is significantly larger compared to the base-run. In order to determine the ability of policy 4 to achieve better quality within a cost-beneficial frame, the benefit-cost ratio is examined. The benefit-cost ratio must be improved as compared to the base-run in order to achieve this goal.

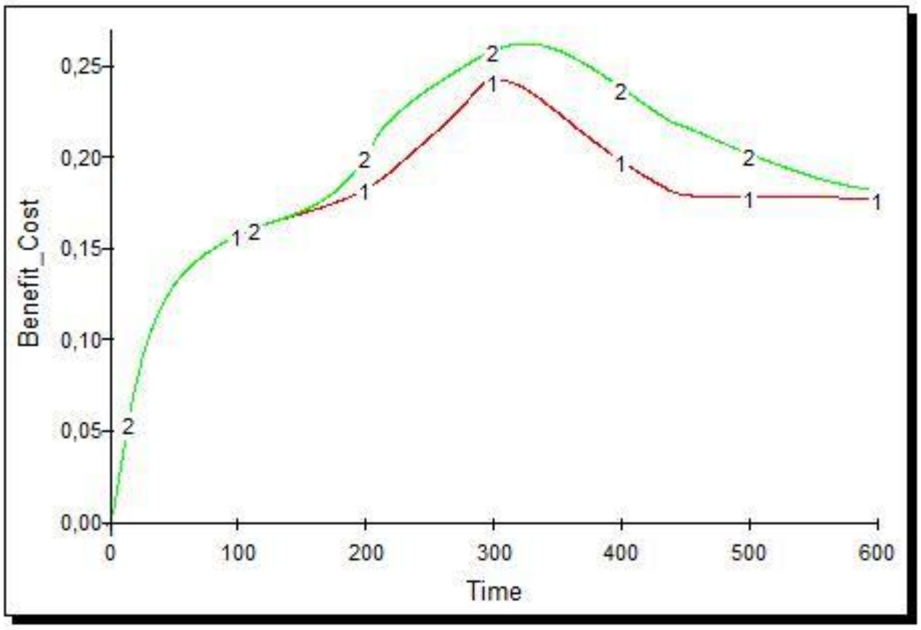


Figure 126. Benefit-cost,
Red line presents the base run and green line presents the policy run

From the graph in figure 126, it is apparent that the benefit-cost ratio for policy 4 yields a positive result compared to the base run. In the beginning the cost-benefit rate is equal, but the inclusion of testing changes the result in favor of policy 4. The increase in detected errors and subsequent rework-rate outweighs the additional costs in man-days. The level of benefit-cost ratio for policy 4 is at all times higher than the base-run after testing and client review has been implemented. This development strengthens the argument found in the literature. For example both Hjertø and Deming stress the importance of including the client (Hjertø, 2003). In modern medium-ranged projects this is a common procedure that can take different forms. My model incorporates the client acceptance test as described by Jones and Bonsignour. (Jones & Bonsignour, 2012: p.320).

The behavioral pattern in this graph reflects the difference in benefit-cost between the base-model and policy 4. The larger peak that occurs in benefit-cost ratio for policy 4 is the result of both a continuous testing effort and the client review. This creates the observable large and sustained peak in the benefit-cost ratio for policy 4. Towards the end of the project, the effort is closing on its goal of zero tasks remaining. Hence, both the detection and correction of errors decreases as fever development tasks are completed, and manpower is transferred to the final systems test. The benefit-cost ratio follows a goal-seeking pattern and declines towards its goal-value at the end of the project.

8.1.3 Comprehensive analysis of synthesized policies

During the course of this chapter I have described some managerial policy-options available for the three main efforts under investigation in my thesis. By altering the distribution of manpower, the model provides vital information on changes in the ability to detect and rework errors. However, as I have demonstrated through my causal loop diagrams the relationships between each effort is interlinked in a wide array of complex connections across subsystems. The ability to introduce policy options isolated for each effort is only the first point of policy-construction. In this part of my analysis, I will combine the four policies together. This combination allows a synthesis of the isolated policies in order to provide managerial decisions regarding all efforts simultaneously, with the effect of each isolated policy at play.

In my comprehensive analysis, I will therefore provide managerial policies regarding all three efforts simultaneously. They will vary in allocation-distribution, and their effect on the software project process is analysed. The aim of my analysis is to reveal how different policy strategies regarding manpower-distribution is beneficial for the overall software quality, and that this quality increase holds positive impact on the overall cost-benefit of the software project.

In order to provide this information, each synthesized policy will be analysed through a set of five graphs. The first two graphs are the reference mode provided earlier in this chapter. We observe the change in detected errors waiting for rework, and the detected errors rework rate. The last three graphs are all related to the benefit-cost ratio and the amount of errors detected and removed from the software product during production. In the original thesis of Abdel-Hamid, the question of cost was investigated as the maximum potential error detection for quality assurance versus the cost in man-days spent on the effort during the course of the software production cycle. Since quality assurance staff diverts effort to production, the question of cost is determined by the cost of man-days for quality assurance and loss of potential software productivity (Abdel-Hamid, 1991: p.203-207). In the original thesis the only defect-removal effort was the quality assurance effort. This has changed in my enhanced model, and it’s therefore necessary to re-analyse the relationship between reworked errors and the cost in man-days. Since testing is now a vital part of defect-removal and the effort has been directly linked with quality assurance through capture re-capture, the potential for both efforts to detect errors are in play. The benefit-cost ratio is calculated through the total amount of errors detected and removed and the amount of man-days expended on the project. It is very important that this relationship is thoroughly analysed. In addition to the graphs, I will report the schedule completion date, which is the total amount of days spent on the project, for each combined policy and policy strategy. This forms the basis for a comparison between original completion time, and the combined policy’s completion time.

From my initial analysis of isolated polices it is apparent that the testing effort holds a significant effect on the defect removal effort. The reference mode shows that providing a different testing-regime held the highest numerical sensitivity in both detected errors and the rework rate. This is supported by the literature referred to in earlier chapters of my paper. According to for example Jones & Bonsignour, the testing effort is the most utilized for defect removal. The importance of testing in software projects is highlighted in the empirical material, as over 99% of all software projects contain testing. In the original thesis of Abdel-Hamid this effort was largely effective-less, as the testing regime only allowed one final systems test.

From the massive data gathered by Jones and Bonsignour it is clear that several stages of testing through the entire software production cycle are the norm. Ghezzi et al. argue that for modern medium ranged software projects, it is impossible to hold only one final system test. Due to these facts, every synthesized policy analysed in this chapter contains the policy regarding the allocation of manpower for testing. Allocation of manpower is a fundamental managerial decision in order to improve the quality of process, and reach a higher quality for the end-product. From the isolated policy presented earlier in regards to testing, the test-design for several testing stages yielded the highest change in model behaviour and quality. Hence, I keep this fundamental policy and combine it with the other policies regarding manpower allocation for rework, quality assurance and the policy considering the client review in order to improve quality to the highest standard possible within cost-beneficial limits.

In this manner, I will be able to utilize the model to present potential policies that allows management and clients to form managerial strategies. These strategies hold the potential to improve overall software quality, and to address this aspect of the software crisis. Better quality provides in the long run satisfied users, and less failed projects. To this day, the industry is still branded by cost-overruns, failures, and high post-production maintenance costs. Some of these symptoms are directly related to poor quality of process and lack of proper testing-routines.

8.1.3.1 Policy 5: Synthesized policy from policies 3 and 1

The first combined policy in this analysis combines the policy-options for policy number 3 and policy 1. Policy 3 concerns the allocation of manpower for testing effort. As described in this chapter the allocation of manpower for testing is governed through the "Fraction Of Effort For System Testing" that effects the manpower allocation for the testing process through change in the allocation policy. In the base model Abdel-Hamid the original base-run is set to a distribution of manpower to testing near the end of the project and during second half of the coding phase of development. This occurs when 80% of all tasks have been completed (Abdel-Hamid, 1991: p.79 and115).

Policy 1 concerns the allocation of manpower to the rework effort. The normal managerial strategy is provided by the base-model of Abdel-Hamid, where the "Desired Rework Delay" is set to 15 days (Abdel-Hamid, 1991: p.75). With this policy it will take 15 days from an error has been detected before it is handled by a staff assigned to the rework-detail. This variable is a constant, so for the entire production cycle 15 days delay is set for all errors detected until they are reworked. These two allocations form the base-run depicted in the graphs with a red line designated as run 1.

In policy 5 strategy 1, the focus is on allocation of manpower for both testing, and rework activities. From the analysis of policy 3 we observed that changes in the allocation for testing yielded high results for error detection, and the ability to remove errors. Allocation of manpower to testing in the earlier stages, yielded in particular large changes in both rates. Hence this is an efficient policy to improve quality. In addition to this change in allocation, the second run contains the effect of an increased allocation of manpower to the rework effort.

The isolated analysis revealed that more manpower allocated for the rework effort allows a more effective defect-removal process. The larger amount of rework-staff is able to correct more errors. The positive effects of learning and experience through the project-processes are higher for this group, and stimulate the rework rate. The negative impacts of schedule pressures, inexperienced workforce, motivational losses and turnovers are less. Positive effects on a small group yields a benefit, but the effect is constrained by the size of the group. I.e. they can work faster but not as fast as a large group with same positive motivation. Negative impacts have more effect, as the reduction in efficiency is enforced by the smaller size of the group. A large group is able to work much more under positive influence due to both motivation and size. Negative impacts do cause a slower efficiency, but the group is larger and can therefor do more work than a small group in the same situation. This provides a more robust rework effort, and they benefit from increased correction-efficiency. Therefore the second run of my model provides an increase in allocation of manpower to both testing and the rework effort.

For the testing process, the policy of starting testing earlier in the production cycle is deployed. The allocation of manpower is initiated in the middle of design phase of development when 20% of all tasks have been accomplished. This provides a testing regime that performs continuous testing, and feeds the information back to the quality assurance effort and the rework effort. The capture-recapture process is active in this policy, allowing tagged errors to be resent for rework as bad fixes from previous rework, and untagged errors to be resent for quality assurance as escaped errors from previous quality assurance. For the rework process, the desired rework delay between the detection of an error to correction initiates is set to 10 days rather than the original 15 days. This provides more allocation of manpower to the rework process, and the rework effort starts earlier.

Policy 5 strategy 2 for this combined policy is to hold an early and continuous testing process after 20% of all tasks completed, but with a smaller rework staff. In this policy-option we investigate the benefit of having more people allocated to the software development and testing processes versus allocation to rework. Since rework-staff are routed from production to rework, this policy analyzes the possible benefit of the reduction in allocated manpower to rework by having a longer rework delay when testing is indeed finding more errors at an early stage. This is because less manpower allocated to rework leads to more manpower remaining for development and testing effort. These two policy strategies are now simulated, and the results are reported below. First each model run is explained through the reference mode. I will provide a description of the stages, and factors that holds an effect on the observed behavior. After this description, I gather the numerical data and compare the two policy strategies through the cumulative detected errors reworked, the cumulative man-days expended in the project and the benefit-cost ratio.

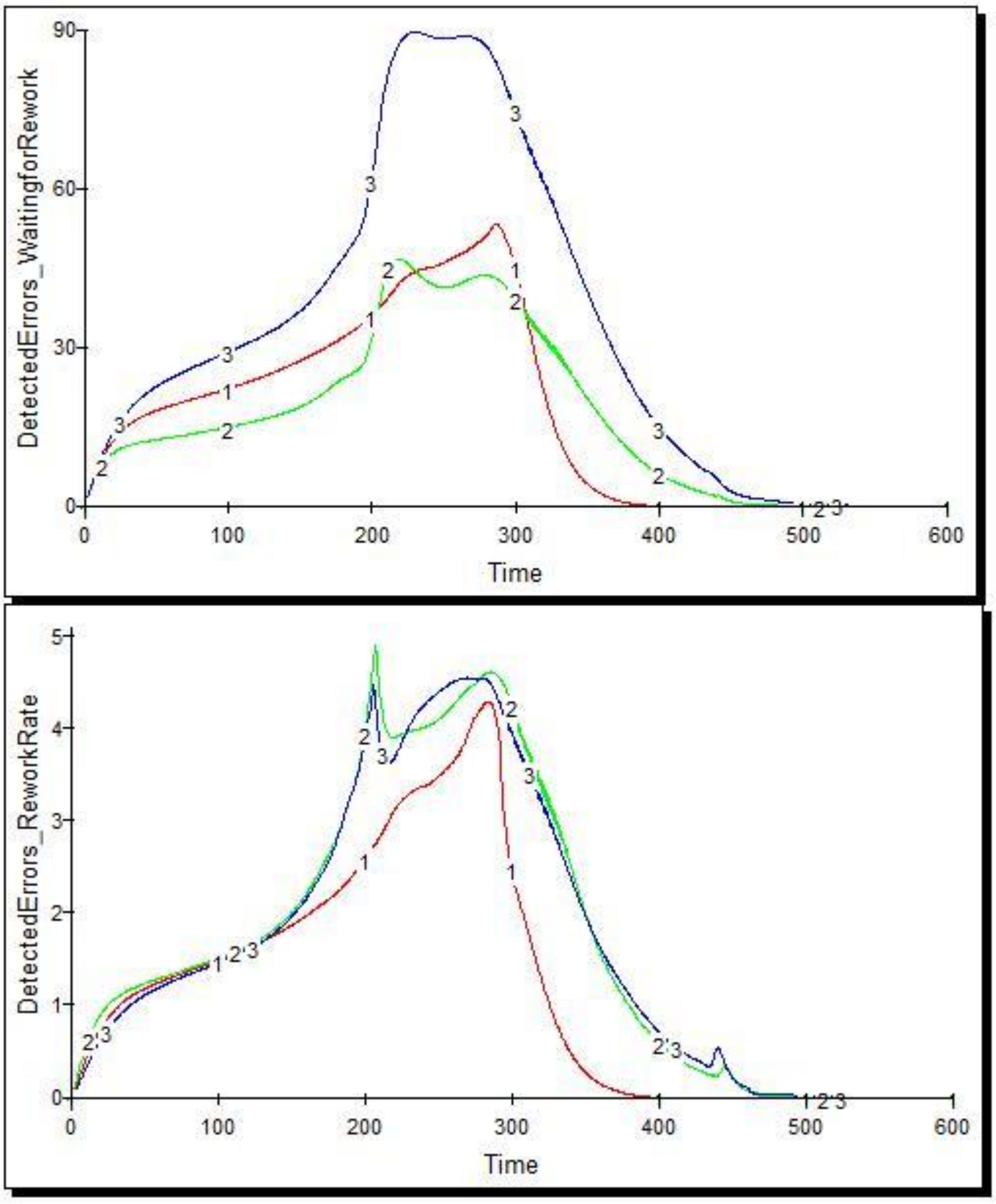


Figure 127. Detected errors waiting for rework and detected errors rework rate, Red line presents the base run, green line presents the strategy1 and blue line presents strategy2

Policy 5 strategy 1

When the software project starts, software is being developed. The tasks that are produced and completed are sent to the quality assurance effort, where errors are detected. While tasks are developed, errors are generated as well. The quality assurance effort detects errors; these detected errors are sent to the rework where they are corrected. In this policy-strategy the desired delay for correction is set to 10 days rather than the original 15 days. In the graph of detected errors waiting for rework, the effect of an increased allocation of manpower for rework through the reduction in desired rework delay is imminent. The rework team starts to correct the detected errors at a high rate, which is observable in the detected errors rework rate. The effort is higher, and the amount of detected errors waiting for rework holds a lower level through the first period compared to the original base-run. The amounts of detected errors do continue to increase steadily through the initial stages of the project. The reason for this increase rests with the rework staffs ability to perform correction, and the number of tasks developed and completed. Initially the rework team starts the process fresh, and eager to complete the correction-tasks. This is a positive motivational factor, which is beneficial for the process. This is observable in the rework rate which starts out with an initial strong increase in errors corrected. However, after a period of time the rework team starts to feel fatigue from the initial boost. Reworking design errors demand concentration, as the errors are fundamental in the software fabric. Hence, the rework rate starts to decrease slightly during a period of time. The second reason for the increase lies in productivity. When the project starts, the first tasks completed are fundamental design-tasks containing complex codes. As the project processes, the software teams move into more rudimentary design tasks. The result is an increase in developed and completed tasks sent for quality assurance. More tasks accomplished translate into more errors being committed and the amount of detected errors increase steadily. Still, the level of increase is lower than the base-run and the total amount of errors waiting for rework lower.

The first fundamental change in the stock of detected errors waiting for rework occurs when project is in the second half of design phase. The reasons for this sudden increase in detected errors waiting for rework are linked to the testing effort. The testing process is activated in the middle of design phase when 20% of all tasks have been completed, and detected errors in testing are fed back to production to get reworked. As a result, the number of detected errors waiting for rework increases sharply. The enhanced testing regime performs functional tests on applications in order to determine their suitability. From the capture-recapture effort, errors have been tagged by the quality assurance effort. The testing efforts identify the rework-considerable errors, and identify them as either tagged or untagged errors. These errors are now returned to either the quality assurance effort or the rework effort. Either way, the errors end up in the stock of detected errors waiting for rework. Compared with the base-run, the increase in detected errors waiting for rework is significantly stronger. The increase in the base-run holds a steady smooth trajectory, while the increase for strategy 1 is a sharp steep increase. This depicts the impact of an early continuous testing-effort.

The response to this increase in errors is visible in the detected errors rework rate. The rework team has been able to recover from the initial impact of the rework effort. Secondly, the project is now situated in the final parts of the design phase. The errors from this phase are simpler to correct, and the rework team has been able to gain valuable knowledge of the inner workings of the software project being produced. With an increase in detected errors waiting for rework, the sense of urgency and schedule pressure cuts slack time, and the rework team puts more actual hours into the correction effort. In addition, this increase in errors waiting for rework signals to management that the total staff-level should be adjusted. More hirees translate to more people available for the rework effort. This constitutes the spike in the rework effort visible in the second half of the design phase of development. The observed rework rate at this point in strategy 1 is higher than the base run, as the number of detected errors requires a stronger rework effort.

This initial boost in rework decreases the amount of errors waiting for rework. In the same time, the project moves from the design phase to the coding phase. Coding errors are fundamentally easier to rework, and therefore the effort needed per average error decreases. The previous increase in the rework rate has also exhausted the rework staff. Fatigue, loss of motivation and turnovers influences the potential rework-rate downwards. Due to schedule pressure, the slack-time was cut and more hours spent on actual rework. We now observe a reduction in the rework rate as a result of these factors. In addition, the lower level of detected errors waiting for rework signals to management that re-adjustments to software development is possible. Hence, more slack-time is allowed and focus is shifted to software development. The rework rate is reduced as a result. This does not occur in the base-run as there was no initial peak due to the testing effort. For the base-run, the rework rate increases as more tasks are being completed at this stage in time.

The amount of errors waiting for rework decreases for a period of time, but then starts to increase during the second half of the coding phase of development. The reason for this second increase lies in the increased completion in tasks. Since the software project is has progressed from design to coding, the software production teams are able to finish more tasks at a higher rate. This leads to an increase in detected errors. In addition more applications are completed for testing, so more errors are fed back to the rework process.

The rework staff is yet again able to increase its productivity as the exhaustion level is low. In addition, the schedule pressure from closing on deadline provides the team with an additional incentive to cut slack time and finish the correction effort. There is a positive motivational factor as well, since the number of tasks remaining is closing to zero. This provides a sense of completion for the rework team, and they work more efficiently as a result. This is observable in the rework rate, as it increases again during the second half of the coding phase of development. At the end of the project, all the tasks have been completed and more manpower is moved from development, quality assurance and rework to testing. The final systems test is the final part of the software project as in the original scheme. There are still tasks that contain errors and needs to be reworked, and they are completed by the end of the project. The detected errors rework rate and the detected errors waiting for rework decreases in a smoothing fashion as people are being transferred from development, quality assurance and rework to testing steadily through this final period. This process is similar to the final stages in the base-run as well. However, due to the larger amount of errors detected, the effort needs more time to complete all errors still waiting for rework. Therefore the decline in strategy 1 is slower than the observed base-run reduction.

Policy 5 Strategy 2

As in strategy 1 the project is initialized, and software is produced. Completed tasks are sent to quality, where errors are detected. While tasks are developed, errors are generated as well. The quality assurance effort detects errors; these detected errors are sent to the rework where they are corrected. In this policy-strategy the “Desired rework delay” for correction is set to 20 days rather than the original 15. For this policy-strategy the reduction of rework-manpower through a longer desired delay for rework, provides more resources available for software development and testing. The effect of such a policy is clear from the initial stages of the software project.

The decision to hold a desired rework delay of 20 days causes the rework effort to start later, and less manpower is allocated to this effort. The result is a sharp increase in detected errors waiting for rework. Quality assurance is detecting errors and sending them for rework, but the smaller staff is less able to rework them at the rate observed in both the original base-policy and policy option 2. In addition, the smaller rework-staff translates to a larger development staff. More development staff equals more tasks accomplished that contain errors.

In addition to the factors mentioned above, the negative effects of exhaustion, turn-over, schedule pressure and loss of motivation hold a greater impact on the rework team. As explained in the introduction to this policy analysis, a smaller staff is more susceptible to negative impacts. Less workers makes the effort constrained by the staff-size, and therefore extra losses to motivation or exhaustion causes a greater impact. In a larger group, losses of motivation and exhaustion would impact the rework-rate, but the larger size would negate some of that effect. A larger group is able to work more under the same negative conditions than a smaller group. As in strategy 1, the rework team's initial correction-rate starts out strong. Due to positive motivation, and a rested crew the initial rework rate increases sharply. However, due to the longer desired rework delay time and smaller staff allocated in this case this initial effort is lower than the base-run. The rework team is also unable to maintain the same amount of rework efficiency for a longer time-period. The effects of exhaustion due to the nature of design errors level off the increase in the rework rate in the same manner as for the base-run and for strategy 1.As a result, there is a larger increase in detected errors waiting for rework through the initial stages of the project. This is clear from the blue line in the detected errors waiting for rework. While the project moves from fundamental design to rudimentary design, more tasks are accomplished and errors generated are then detected by quality assurance effort. The difference in detection level is visible in the graph for detected errors waiting for rework. Compared to the base-run the increase is substantially higher, at the same time the rework rate is close to the initial base-run level.

The project moves along, and the testing effort commences in the middle of design phase of development when 20% of all tasks have been developed and completed. This results in an additional sharp increase in the detected errors waiting for rework. This effect is more dramatic for strategy 2 than strategy 1, since the smaller rework staff is unable to accommodate this increase in errors to the same degree as the larger rework staff. Secondly, less manpower allocated to rework leads to more manpower for production activities and testing. More tasks are completed, and more errors are detected in testing and fed back to quality assurance and rework to get corrected. It is therefore very interesting to see the effect of the testing effort on the rework rate. Since the staff has been able to recover from the initial exhaustion level, the rework rate increases in tandem with the increase in detected errors waiting for rework. Due to the sudden increase in errors provided by testing, management must adjust the total staff-size in order to accommodate this increase. More hirees equals more manpower for rework and the effort shows the same sharp increase in the rework-rate as for strategy 1. This is an interesting observation, as the rework rate succeeds by far the base-line. When testing is in effect, and the information is fed back to quality assurance and rework, and the true nature of the rework becomes is apparent. This leads to an increase in the rework effort trough schedule pressure, cut of slack time and an increase in total manpower size that allows more errors to be reworked despite a smaller ratio of staff compared to the base model.

The initial boost in rework-rate exhausts the rework-team as the negative effects of stress, schedule pressure and motivational losses kicks in. These effects are more severe for a smaller team, and we can see from the rework-effort graph that the dip in rework rate falls lower in strategy 2 than for strategy 1. However, the amount of detected errors that awaits rework is much higher for strategy 2 compared to strategy 1. The result is an additional increase in the rework effort that exceeds the rework-rate in strategy1 and the base run. At this point in time, all tasks have been completed. The development and rework teams are transferred from development and rework to testing, and the remaining tasks that need rework are done completed by the end of the project. Initially the reduction in detected errors waiting for rework in strategy 2 is faster than for strategy 1. This is caused by the smaller amount of overall detected errors for strategy 2. This concludes the simulation for policy5 strategy 2.

Analysis report for completion time, detected errors reworked, man-days expended and benefit-cost ratio

Simulation run	Completion date
Base-Run	438 days
Policy 5 strategy 1	516 days
Policy 5 strategy 2	514 days

Figure 128. Project completion time for base run and policy strategies runs

The first table I present is the completion time for all the simulation runs. In the base-run, the software project was completed after 438 days. For strategy 1 the software project was finished after 516 days, and for strategy 2 the completion time was 514 days. This provides an indication of increase in production time when the testing process is continuous. There are two main reasons for this increase. The first reason lies in manpower-allocation. When manpower is allocated to both rework and testing, there is less manpower available for the software development effort. As a result less software is produced per day, and the project schedule must be lengthened. The second reason lies in the detection of errors. With continuous testing, and capture re-capture the potential for error-detection is much higher than for quality assurance alone as in the original model. This is identified by numerical data provided by Jones and Bonsignour, where 80-90% of all errors can be detected by utilizing an effective testing system alongside a capture-recapture system (Jones & Bonsignour, 2012). Hence, more errors need to be reworked before the system can initiate the final systems test.

The next table that I present reports the total amount of errors detected and reworked during the course of the software project. The aim is to find the best suitable policies for defect-removal, providing increased quality of both process and end-product. The total amount of errors found and removed, is the benefit for each policy strategy. The table below contains the total amount of errors detected and corrected.

Policy simulation	Defect removal
Base run	709,40 errors
Policy 5 strategy 1	1003.84 errors
Policy 5 strategy 2	986 errors

Figure 129. Project defect removal for base run and policy strategies runs

For both strategies 1 and 2, the total amount of defects removed is a clear improvement compared to the base-model. These results are in line with the development observed through the reference graphs presented in the previous part of the analysis in figure 127. The addition of a continuous testing system and the capture re-capture effort allows more errors to be detected and reworked. As a result, both strategies deploying a shorter desired rework delay and a longer desired rework delay benefits in terms of detected errors. The difference is not large, but strategy 1 with a shorter desired rework delay, is able to remove 17,84 additional errors compared to strategy 2.

The next table that is displayed here contains the cost-side to the two policy strategies compared to the base-run. The costs for the software project are measured in man-day expenditure. To determine the benefit of a policy strategy, it is important to consider the costs for the project as a whole. Detection of errors may come with an additional price that outweighs the benefit. The next table reports the total cost in man-days for each policy strategy and base-run.

Policy simulation	Man-day expenditure
Base run	3878,70 Man-days
Policy strategy 1	5507,77 Man-days
Policy strategy 2	5408.03 Man-days

Figure 130. Project man-day expenditure for base run and policy strategies runs

In terms of man-days the table above in figure 130 shows a significant increase in the man-day expenditure for both policy strategies compared to the base-run. This is consistent with the completion time, and manpower allocation in these two strategies. For strategy 1 the decrease of desired rework delay fosters a larger rework-effort at an earlier point in time. Due to the increase in errors detected by the testing-effort, management is continuously adding more effort to correction of errors. With the continuous testing effort more manpower is allocated to defect removal, hence the software development rate is lower for the project over time compared to the base run. The increase in errors detected leads to an increase in the total manpower-level for the project leading to higher expenditures of man-days.

For strategy 2, the same effect of continuous testing is causing the completion-time to end significantly later than the base-run. Even though the strategy incorporates a longer desired rework delay and therefore a smaller rework staff, but the additional allocation to testing leads to a higher level of detected errors waiting for rework. Therefore more manpower is deployed for defect removal, and man-day expenditure increases. However, compared to strategy 1 the total man-days expended is lower by 99.74 Man-days. Both these strategies improve the error detection level compared to the base model, but expend significantly more resources in terms of man-days.

The question that now remains is the benefit of these two policies compared to the original base-run. This is provided through the three final graphs depicted below. They are the cumulative detected errors reworked that presents the benefit of the project, the cumulative man-days expanded that presents the cost of the project and the benefit-cost ratio. For these policies to have a positive effect, the detection and rework of errors must outweigh the additional costs in man-days. These two graphs are able to provide the dynamic development in expenditures over time. The tables are only able to provide the static end-result. With these graphs we are able to observe the changes over time, and further explain the results. The graphs are presented below in figure 131 :

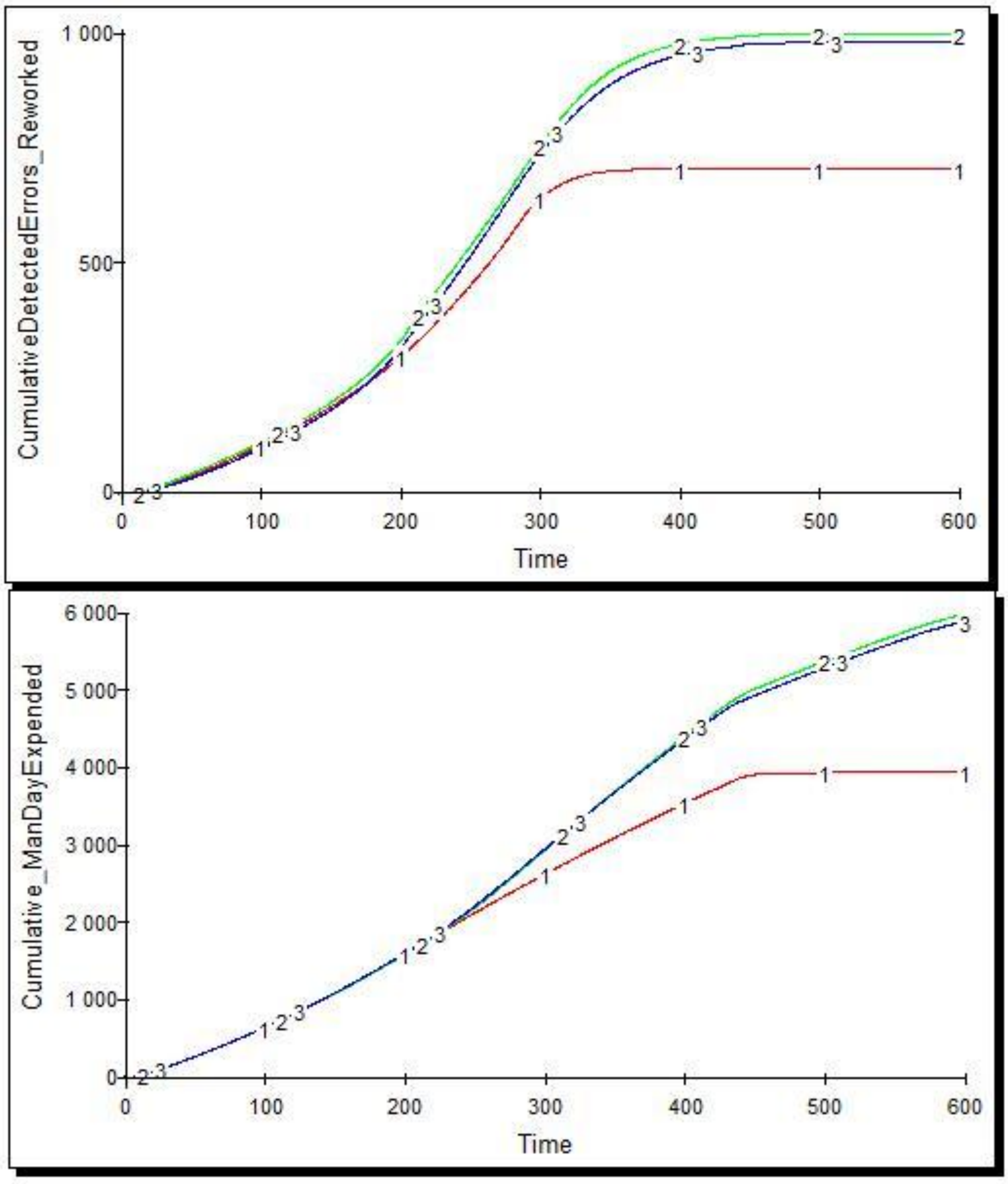


Figure 131. Cumulative detected errors reworked and cumulative man-day expended, Red line presents the base run, green line presents the strategy1 and blue line presents strategy2

From both these graphs above in figure 131 we observe the cumulative values of the reworked detected errors, and the cumulative man-day expended. First graph shows the cumulative detected errors reworked that presents the benefit of the project. From this graph we observe that in both our policy strategies, the quality assurance, rework and testing teams were able to detect and rework a much higher amount of errors than in the base model. The departure from the base-run occurs when the testing-system is initialized, and the detection of errors increase substantially. This result is not unexpected, since the isolated policy-analysis explained earlier in this chapter regarding manpower allocation for testing indicated that the most important source for error detection is the enhanced testing regime.

The size of the rework-staff holds less impact since they provide the error correction, and not detection itself. Although it is important to note, that the larger rework-staff in strategy 1 were able to rework more errors than in strategy 2. The reason for this lies in an effect of schedule pressure on testing described in chapter six. When there are several errors waiting rework, the schedule pressure for the entire project increases. Due to the smaller rework staff of strategy 2, the amount of detected errors awaiting rework was much higher than for the base run. Under such circumstances, the schedule pressure leads to managerial re-evaluation of standards for significant errors. For example, if the original standard dictated that a piece of software should pass five criteria any piece of software failing this test would be considered significant. Under schedule pressure, pieces of software that pass four out of five criteria constitute a challenge. In order to complete on time, it is possible to lower the standard to four out of five to pass the test. The piece of software would still be operational, but the overall quality would suffer as a result. Unfortunately, as described in chapter six, requirement volatility occurs under schedule pressure.

The second graph shows the cumulative man-days expended. This graph supports the completion-time table provided earlier. For both policy options 1 and 2 the cumulative man-days expended are by far higher than the base run. Due to the increased amount of detected errors, more manpower is deployed to all efforts for a longer period of time. The testing-regime is the prime engine for this development. This is visible in the graph, as the development for policy option1 and 2 do not deviate from the base-run until the testing effort is activated. In sum, for both policy options we were able to detect and rework more errors providing better end-product quality. In the same time, we expanded more man-days that presents the cost of the project and delivered the project at a later date.

Just as we observed for the isolated analyses in this chapter, the development in both graphs regarding the cumulative detected errors waiting for rework, and the cumulative man-day expenditure display a characteristic pattern through the software development stage. This pattern is consistent with the base run, and follows the same logic. As the project is advancing, the defect removal effort starts to rework errors. As a result, the cumulative errors reworked increases. Due to factors like increased production, experience, and progress from design and coding the increase takes form of exponential growth. The same phenomenon occurs in the graph for man-days expended. The increase in activity increases the level of man-day expenditure.

After a period of time, the graphs change behavior-pattern towards the end of the project. There are fewer tasks that need to be completed, as the project is moving towards its completion. The goal for the project is to reach zero tasks remaining, and the behavior of the graphs depicts a goal-seeking behavior. The closer the project gets to the completion the graph levels off towards the cumulative end-level for both detected errors and man-day expenditure. The result is a graph that follows an S-shaped development-pattern between initial exponential growth, and goal-seeking behavior. This is a characteristic behavior for these two graphs in this model. The behavior is the same between policy and base-run, but the level is different due to the combined policy strategies and manpower allocation.

The final graph in this combined policy analysis reveals the benefit from error correction with the costs of expanded man-days.

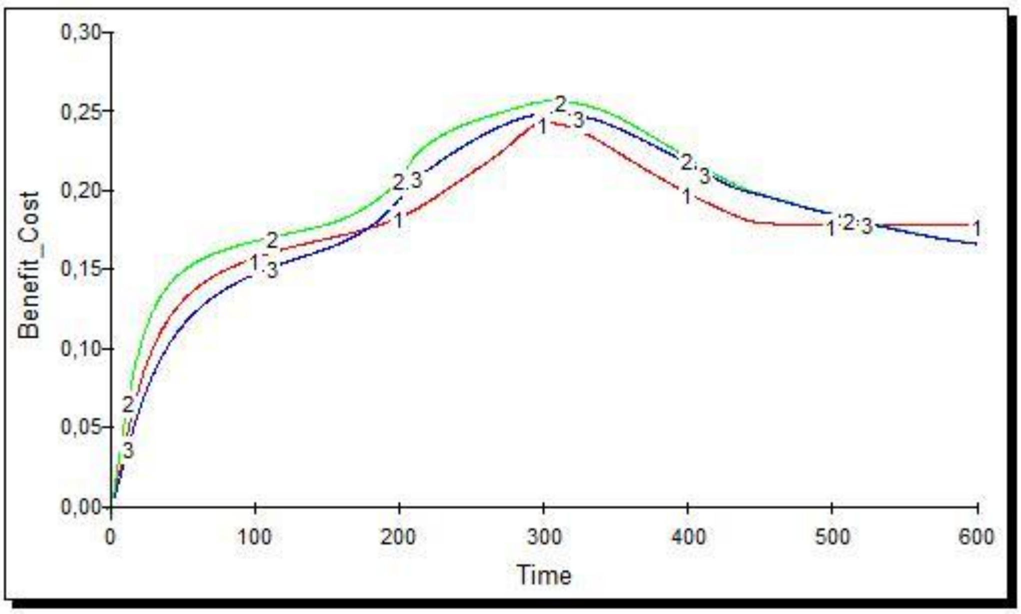


Figure 132. Benefit-cost,
Red line presents the base run, green line presents the strategy1 and blue line presents strategy2

From the graph we observe the development in benefit-cost ratio for all three simulations. In order for our policy to provide better software quality within a cost-effective frame, the benefit-cost ratio must be higher than the base-model's yield. For strategy 1 we observe that the shorter delay-time activates the rework-effort earlier and higher by allocation more manpower to rework effort. This yields a defect-removal effort that holds a higher benefit-cost ratio compared to the base run. The removal of errors early in the process is beneficial. This is strengthened by the benefit- cost ratio for policy strategy 2. Due to the smaller rework staff and the longer delay, the initial effort to rework detected errors yield a smaller benefit-cost ratio for strategy 2 compared to the base run. However, as the impact of errors detected by testing increases the rework effort through managerial allocation, the benefit-cost ratio surpasses the base-run and continues to yield a higher rate. The effect of testing is visible for both policy strategies, as the sharp increase in benefit-cost, earlier in the project and staying in this higher pick amount for a longer period, is not visible in the base run. The base-run has only the final systems tests, and do not benefit from the continuous testing effort.

Jones and Bonsignour's analysis of software projects concludes that increased software quality provides more benefits than costs to the software project (Jones& Bonsignour, 2012: p.433-544). This seems also to be the case for my policies. In sum it is apparent that the benefit of having a continuous testing regime outweighs the extra costs. The most beneficial policy is strategy 1 that holds the highest benefit-cost ratio through the entire project. The difference in completion time between the two strategies are only 2 days, so there are no big gains of running a smaller rework staff in regards to completion time. In sum, this analysis proves the significant role of testing as the main provider for increased software quality and increases the benefit-cost ratio.

Finally I will make two observations for policy 5 strategy 1 and strategy 2. In the detected errors waiting for rework graph in figure 127, it may initially look like the amount of errors detected waiting for rework for strategy 1 is lower than strategy 2. This is a common misperception that is important to avoid. The reason for the lower amount of detected errors waiting for rework lies in the larger rework staff's ability to correct errors at a shorter amount of time. When we look at the cumulative detected errors reworked, this fact becomes evident.

The second observation is linked to strategy 2. Even though the smaller rework team was able to correct an increased amount of errors compared to the base-run there was an issue for the rework rate. The effort needed to increase significantly at two occasions, causing additional strain and exhaustion to the rework team. Such increase in the effort needed is not without risks. If the exhaustion level is large enough, such a policy could lead to a higher rate of turnover, causing additional cost-problems for the software project.

8.1.3.2 Policy 6: Synthesis of policy 3 and 2

The combined effect of policy 3 and policy 2 is the core of combined policy 6. The focus in this policy-strategy is the effect of the testing effort through the allocation of manpower for testing, and the allocation of manpower for the quality assurance effort. The allocation for testing is controlled trough the variable “fraction of effort for systems testing”. The allocation of manpower to the quality assurance effort is governed by the variable “planned fraction of manpower for quality assurance”. In this synthesised policy, the base run is shown as the red line marked with “1”. The green line marked with a “2” is the policy-option simulation.

The base-run for this policy allocates the original manpower for testing when 80% of all tasks are completed. This allocation dictates a single test-regime, where no feedback from testing enters the system. In addition, the planned fraction for quality assurance is set to 15% of daily manpower after training. The quality assurance manpower is derived from the total work-stock before the separation into production and defect-removal. Hence, after training of personnel has been accomplished, a given fraction of manpower is assigned to quality assurance. The quality assurance sector provides bureaucratic functions such as walk-troughs, formal interviews and project documentation. In addition the quality assurance effort performs micro-level testing, and identifies and tag errors in the capture-recapture procedure.

For the combined policy I wish to analyse the relationship between a continuous testing effort and a higher fraction of manpower allocated to quality assurance. From the isolated policy-analysis of policy 1 we observed that the introduction of a continuous testing regime held a large impact on the total amount of errors detected and reworked. In addition, the isolated policy-analysis of policy 2 concerning the quality assurance effort revealed that this effort is crucial for improving the quality of both process and end-product. Jones and Bonsignour suggest that a project able to reap the benefit of a tandem testing/QA-system will be able to effectively remove a higher range of errors before production has ended (Jones & Bonsignour, 2012). Hence, for the combined policy 6 the second run incorporates a testing-regime where manpower is allocated to testing effort early in the project and in the middle of design phase of production when 20% of all tasks have been accomplished. For quality assurance, the planned fraction for quality assurance is set to 20% of the total daily manpower after training. The simulation-results are depicted in the graphs below in figure 131.

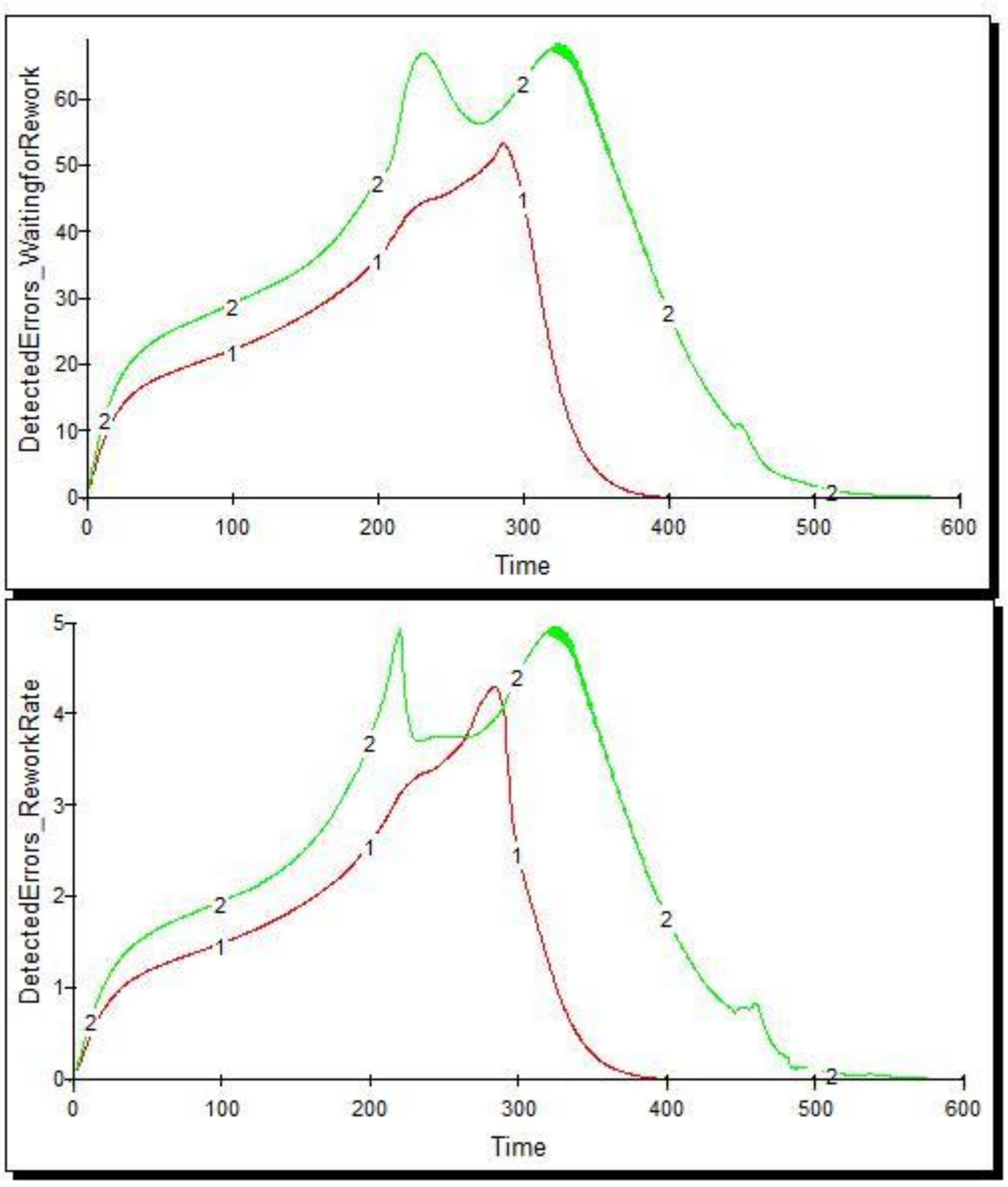


Figure 133. Detected errors waiting for rework and detected errors rework rate, Red line presents the base run and green line presents the policy run

In this policy-strategy we observe that having an increased quality assurance staff, provide an immediate impact on the errors detected at the early stage of software production. When the software project starts, the main tasks constructed and completed are design tasks that include design errors. This indicates that error density is low, and it takes an effort for quality assurance to detect and find such errors. The governing factor for quality assurance is their potential to detect errors. As explained by Abdel-Hamid, the potential for error detection is directly influenced by the amount of personnel allocated for the quality assurance effort. In the combined policy 6 the initial level of people allocated to the quality assurance staff is higher. We observe in the stock of detected errors a sharp increase in detected errors waiting for rework, as the larger quality assurance team detects errors at a higher rate.

This increase in detection holds a significant influence on the rework rate as well. From the second graph we observe that the initial level of rework holds a strong and high rate. There are two factors playing in at this point in time, explaining the high rework rate. From the combined policy 5 we observed that the initial rework rate is high due to a rested and eager rework-staff, and that positive motivational factors are present at the start-up of a new project. In addition, the early increase in error detection sends a signal for management that the rework effort must be efficient. With an early sharp increase in detected errors waiting for rework the rework effort is strengthened by the managerial decision of allocation. This is an interesting observation, as the initial rework level is able to maintain over time a higher rework rate compared both to the normal initial run and the rework-rate observed in the combined policy 5. As observed earlier, the negative impacts on the rework rate such as fatigue, stress and loss of motivation decreases the rework rate for a period of time, and the rework rate holds a slower but steady increase. The rework-rate is initially higher than the base run due to this increase in detected errors waiting for rework.

The detected errors waiting for rework also decline in increase in a similar fashion to the observed rework rate. The reasons for this relatively lower increase in detected errors waiting for rework are mirroring the rework rate factors. The quality assurance staff is susceptible to the same motivational factors that influence the rework staff. Initially the quality assurance effort starts out in a strong fashion due to the positive motivational factors influencing the effort. The quality assurance team is rested, and there is an aura of excitement due to the initiation of a new software project. Hence, their ability to detect errors is high. Over time however, fatigue and pressure lowers the quality assurance team's potential for detecting errors. The initial errors are design errors, and the error density is low. This translates to a situation where quality assurance staff must exert more effort per average error in order to detect and identify them.

The second effect in play is the allocation of resources and number of tasks completed. Since the increase in detected errors occurs at a high rate, management allocate more for the rework effort as a response. We have a policy where more personnel have been transferred to the quality assurance effort, transferring potential resources from production to defect removal. Quality assurance teams are only concerned with the bureaucratic functions of production, defect detection, and capture re-capture. They do not participate in production. This situation in addition to an increase in the rework-staff that diverts more manpower for defect removal rather than production, leads to less tasks produced. With fewer tasks to investigate, fewer errors are found as a logical consequence.

The third factor is learning and experience. Since the project is moving from fundamental design to more rudimentary design, the production team has increased knowledge of the software they are creating. When the blueprint has been completed and rework has removed the initial errors, production is able to produce tasks that contain fewer errors.

The fourth factor for this declining increase is overheads that delay the efficiency of quality assurance. The quality assurance effort demands a system if communication with the production team. This is obtained through reviews, formal interviews and meetings. Boehm observes that every chain of communication between quality assurance and production leads to delays. These delays from the communication and bureaucratic function of quality assurance reduce the productivity-level of the software production side. This leads to fewer tasks accomplished and less errors to detect. For the quality assurance staff, the delay from communication is lowering the potential for detection. This translates to an additional declining increase in the error detection rate, and consequently on the amounts of detected errors waiting for rework.

However, as we observe in the graph above in figure 133, the level of detected errors is significantly higher than the base-model. The initial effort of the quality assurance staff due to the increase in staff-size increases the detection rate well above the original rate. In addition, the same negative factors influencing the quality assurance team for combined policy 6, influences the quality assurance staff in the base-run as well. The larger quality assurance staff of combined policy 6 is able to cope with these factors better than the base-model due to the strength in size.

After a time-period with this relative smooth increase in detected errors waiting for rework, the effect of testing enters into play. This takes place in the middle of the design phase of development when 20% of all tasks have been completed, and the errors detected are fed back to either quality assurance or rework. This facilitates another sharp increase in detected errors waiting for rework, and we observe a continuous increase that results in a peak near to the end of design phase of development.

In addition to the effects of testing, the quality assurance staff has been able to recover from the initial exhaustion. In addition, the quality assurance staff and production staff have been able to incorporate a more effective communication-level. Experience, personal connections between staff-members, and predictable routines allows for smoother communication and fewer time-losses for communication. This increases the quality assurance staff's error detection potential, and more errors are being detected. In this time-period the project has moved from design to coding, allowing more tasks to be completed at a higher rate from the development-side. More tasks completed leads to more errors being committed. Fundamental coding tasks still require more effort than the final and simpler coding tasks, so errors do occur at a higher rate. Since the rate is higher, the error density increases as well. Higher error density increases the quality assurance staff's potential for detection. Higher error density reveals error-patterns that reveal chains of errors.

From the rework side we observe the same increase in rework rate. The initial exhaustion of the rework staff has diminished, and they are able to rework more errors. Since these errors are more coding-errors and simpler to rework, more errors are reworked by the same amount of rework-staff. With the increase in detected errors waiting for rework supplied by the testing-effort and the increased quality assurance detection potential, management is given a signal for an increased rework-effort. The amount of detected errors waiting for rework increases in such a manner that schedule pressure increases. Slack-time is being cut, and more effort is targeting the defect removal effort. This is observable in the rework rate, as the effort increases rapidly and peaks at the end of the design-phase.

Compared to the base run, the increase in both graphs is significantly higher and at a steeper trajectory for combined policy 6. The continuous testing-effort is detecting errors that previously escaped detection in quality assurance, or badly fixed errors from previous rework. These errors are now fed back to the defect removal effort, and the amount of detected errors waiting for rework increases drastically. This peak is observable in the base-run but it is gentler in trajectory. In the base-run the peak is caused by the increase in tasks completed. Experience, the easier nature of coding tasks and increased productivity due to learning. The same effect is increasing the detected errors waiting for rework in combined policy 6, but the effect is added with the effect of the testing-effort. The managerial response to the increase in detected errors waiting for rework is an increase in the rework effort. Slack-time is cut, more hours spent on actual correction of errors and the overall manpower on the project is adjusted upwards. This is observable in graph in figure 133 where the rework-rate increases significantly compared to the base-run.

After this initial increase and high peak in detected errors waiting for rework, we observe a change in behaviour that is unlike that of both the base-run and the observed behaviour of combined policy 5. For both the stock of detected errors waiting for rework, and the detected errors rework rate there is a drastic decrease. Initially from the stock of detected errors it looks like the increased rework effort is able to rework and remove more errors than quality assurance and testing is able to find more errors. However, when we look at the rework rate its apparent that this is not the case. Both stock and rate is decreasing at the same time. The reason for this behaviour lies in the policies we are applying here, and the production of software.

In combined policy 6 we have allocated more manpower to both quality assurance and testing at an earlier stage. This provides a quality-focused strategy. By allocating more manpower to both defect-removal efforts, less manpower is available for development. With this situation in mind, the quality assurance effort immediately starts to detect and identify more errors than in base-run one. This demands an increase in the rework-rate diverting more manpower from development to defect removal. This situation continues, until the testing effort starts. By combining both the increased quality assurance detection and testing detection, the stock of errors increase sharply. Management readjust the rework workforce size and allocation to accommodate the increase in detected errors waiting for rework. This results in an increase in schedule pressure that is substantial.

The more detected errors waiting for rework, the more man-days perceived still needed by management increases. From my CLDs provided in chapter 4 I revealed that this situation increases the schedule pressures significantly. From the empirical data of Boehm and the empirical descriptions from Abdel-Hamid, this situation leads to a managerial decision regarding quality assurance. When schedule pressure is high and production is low, Quality Assurance is suspended for a period, and the rework teams are downsized and put back into development to close the gap between scheduled delivery time and actual delivery time (Abdel-Hamid, 1991: pp.71-72). This is the effect that enters for combined policy 6. Observe that the detected errors waiting for rework do decline smoothly for a period of time while the rework-rate is held at a fixed level. This behaviour indicates that quality assurance has been suspended, and no new errors are detected and waiting for rework. From the rework-rate graph we see that the rate is lower than the base-run rate for a period of time.

The result of the suspension of quality assurance and re-adjustment for software development is observable in both graphs for combined policy 6 in figure 133. There is a drop that resembles a valley in the graph that is not visible in the base-run. Since quality assurance is suspended, the level of detected errors waiting for rework diminishes. Hence, the rework-effort is able to remove errors without new errors being detected by quality assurance. This signals for management that the rework-effort can be re-adjusted to favour software development. For both graphs the rate drops in a similar manner. In the base-model, this situation did not occur. The reason is that the extra level of detected errors causing quality assurance to be suspended is not occurring in the base-model. The sharp increase in exhaustion of the rework-effort is largely due to the combination of continuous testing and enlarged quality assurance staff. Without this factor, the need for suspending quality assurance due to a perceived gap in completion and strong schedule pressure is not present in the base-model.

During the period of suspended quality assurance, the increase in software development allows more tasks to be completed, and the stock of detected errors decreases. Since quality assurance has been suspended, the completed tasks are sent directly to testing. In addition, the increase productivity due to learning and experience enables the development team to finish tasks at a higher rate. The project is now deep into the coding effort, and the simpler coding tasks are finished at a higher rate. This increase in production reduces the perceived man-days still needed, and schedule pressure is lowered. The project is also close to complete the perceived tasks remaining, since the quality assurance effort has not been active. These factors allow management to re-initiate the quality assurance effort. The effect of this restart is imminent. Since the quality assurance effort was suspended, all errors detected by testing were untagged errors. Since quality assurance also handles the tagging of errors in the “capture-recapture” scheme, suspending this effort leads to untagged errors. These are now transferred to quality assurance and identified. In addition, since more tasks are completed the quality assurance effort detects and tags more errors as well. The graph shows how the combined effect of testing and quality assurance again increases the detected errors waiting for rework to reach the second pick. We also observe that the effort has been delayed, and another peak occurs at a later stage than in the base run.

For the rework rate, we see that the increase in detected errors waiting for rework after restarting the quality assurance effort leads to a re-allocation of manpower for the rework effort. The rework rate increases in response to the level of detected errors waiting for rework. At this point in time, the software production tasks have been completed. Manpower is now moved from quality assurance to the final systems test. The same move of manpower occurs for the rework-team. This constitutes the explanation for the slow decline in detected errors waiting for rework and the rework rate. The rework staff is slowly moved to system test, and the remaining staff handles the remaining errors that needs to be reworked. This concludes the simulation of combined policy 6.

<u>**Analysis report for completion time, detected errors reworked, man-days expended and benefit-cost ratio**</u>

The first table I present below is the completion-time for both base-run and the combined policy 6.

Simulation run	**Completion Time**
Base-Run	438 Days
Combined Policy 6	551 Days

Figure 134. Project completion time for base run and policy run

In the table above, I report the completion time for both simulations. In the base-run we had one system-test at the end of the project after 80% of all tasks have been finished, and an allocation of manpower to quality assurance consisting of 15% of all manpower after training. In combined policy 6 we combined the effect of continuous testing at an earlier stage in the project when 20% of all tasks were completed and with a quality assurance manpower level of 20% of all daily manpower after training.

The difference in completion-time is expected, due to the factors and behaviours observed in the graphs above. Continuous testing and an increase in quality assurance provide a longer completion time. There are several reasons for this difference. The first reason lies in the total amount of errors detected. With a higher level of quality assurance staff, the potential to detect errors increased. The stock of detected errors waiting for rework increased in a manner that surpassed the base-run with a clear margin. The additional effect of continuous testing pushed the amount of detected errors even further. At the second half of the design phase of development after 20% of all tasks are finished, we observed a clear peak in detected errors waiting for rework. Hence, the amount of errors detected was much higher than for the base-run. The increase in detected errors yielded an increase in the rework rate. More manpower allocated for rework lead to a decrease in software development because of less manpower remain for development effort and more days were spent to complete the project.

The second factor stems from the schedule pressure that suspended quality assurance for a period of time. This delayed the completion of the quality assurance effort. In addition, the tasks finished under suspended quality assurance where directly sent for testing. This provided a back-log of untagged errors that needed to be addressed by quality assurance when the effort was reinitialized. This caused an additional delay, with a peak of detected errors waiting for rework after the original base-run had completed its number of tasks.

The third reason for delay stems from more allocation of manpower to quality assurance leading to less manpower remaining for production activities, and tasks development rate decreases in circumstances. The fourth reason is communication overheads and exhaustion-factors. More quality assurance effort leads to more communication. These communication procedures take time to incorporate effectively, and lead to initial delays. Also with more errors detected, exhaustion factors for both rework and quality assurance lead to a loss of productivity. These four factors all contributed to the increase in time spent on the project.

The difference in completion time between the base-run and the combined policy 6 is 113 days.

The second report concerns the total amount of errors that have been detected and corrected through the software production cycle. For my policies to yield a better software quality product, the total number of errors removed from the software must be higher than the original base run. The table below in figure 135 contains the total amount of errors detected and removed. This factor is the benefit side of the project.

Policy simulation	**Defect removal**
Base run	709,40 errors
Policy 6	1253.50 errors

Figure 135. Project defect removal for base run and policy run

Comparing the results of defect-removal between base-run and combined policy 6 reveals a significant difference between them. The base-run was able to detect and remove 709,40 errors, while for combined policy 6 the increased quality assurance staff and continuous testing lead to a removal of 1253.50 errors. This difference in result strengthens the argument of Jones and Bonsignour that testing and quality assurance are complimentary processes. The increased error detection potential, combined with capture re-capture and continuous testing provide a significant increase in errors removed from the end product. Testing and Quality assurance provide error detection on the micro and meso-level. Combine these two efforts and the project will benefit greatly in regards to defect-removal (Jones & Bonsignour, 2012), as observed in the table above in figure 135. By detecting and capturing more errors, the rework-effort is increased as a result. This provides an increase in the end-software quality.

The next element that is analyzed here is the man-days spent on the project. The project's cost is measured in the total amount of man-days expended on the project as a whole. For any policy to be cost-efficient, the effort of defect removal must hold a higher benefit than the additional cost of redistribution of manpower from development to the correction effort. The table below in figure 136 contains the man-days expended in combined policy 6 and the base-run. This provides a basis for comparison of costs between the different allocation-patterns.

Policy simulation	Man-day expenditure
Base run	3878,70 Man-days
Policy 6	5639,49 Man-days

Figure 136. Project man-day expenditure for base run and policy run

The results reported in the table above are consistent with my analysis so far. The man-days spent in combined policy 6 are significantly higher than for the base-run. In combined policy 6 a significant amount of manpower is allocated to quality assurance and testing. This factor leads to a lower potential software development rate, and the project needs more time to finish the production cycle. The second element that explains this increase in costs is the effect of detected errors on the rework-effort. When more errors are picked up at an earlier stage, the rework effort is increased to match the level of detected errors waiting for rework. Hence, management is re-adjusting more manpower and effort to the defect-removal side of the software project. This lowers the potential software development rate even further, and schedule pressures ensue. The rework effort peaks at a high rate, and exhaustion factors kick in. The result is a situation where management perceives the project to be severely behind schedule. The remedy for the situation is a suspension of all quality assurance procedures. Manpower is re-routed to software development. This causes a delay in production, and the total amount of manpower is adjusted upwards by management.

Since the quality assurance effort has been suspended, the errors that are found in testing are automatically un-tagged errors. Hence, a back-log of untagged errors awaits identification by the quality assurance team. As a result, when the quality assurance is reinitiated the level of detected errors waiting for rework increases again and causes a second significant peak. More man-days are expended as a result.

This chain of events is not occurring in the base-run, and the effort finishes earlier than combined policy 6. The total amount of man-days spent in the base-model are 1760,79 less than combined policy 6. In sum, the cost level is significantly higher for combined policy 6. However, the ability to correct errors was also significantly improved.

The question that needs to be analyzed now is the benefits of increased defect removal versus increase in man-day costs. In order to analyze this properly I present three graphs below. The first two graphs in figure 137 depict the development of cumulative errors reworked, and the cumulative man-days expended on the project. These graphs are able to determine the dynamic development of both cost and benefit through the software production cycle. The last graph in figure 138 depicts the benefit-cost ratio.

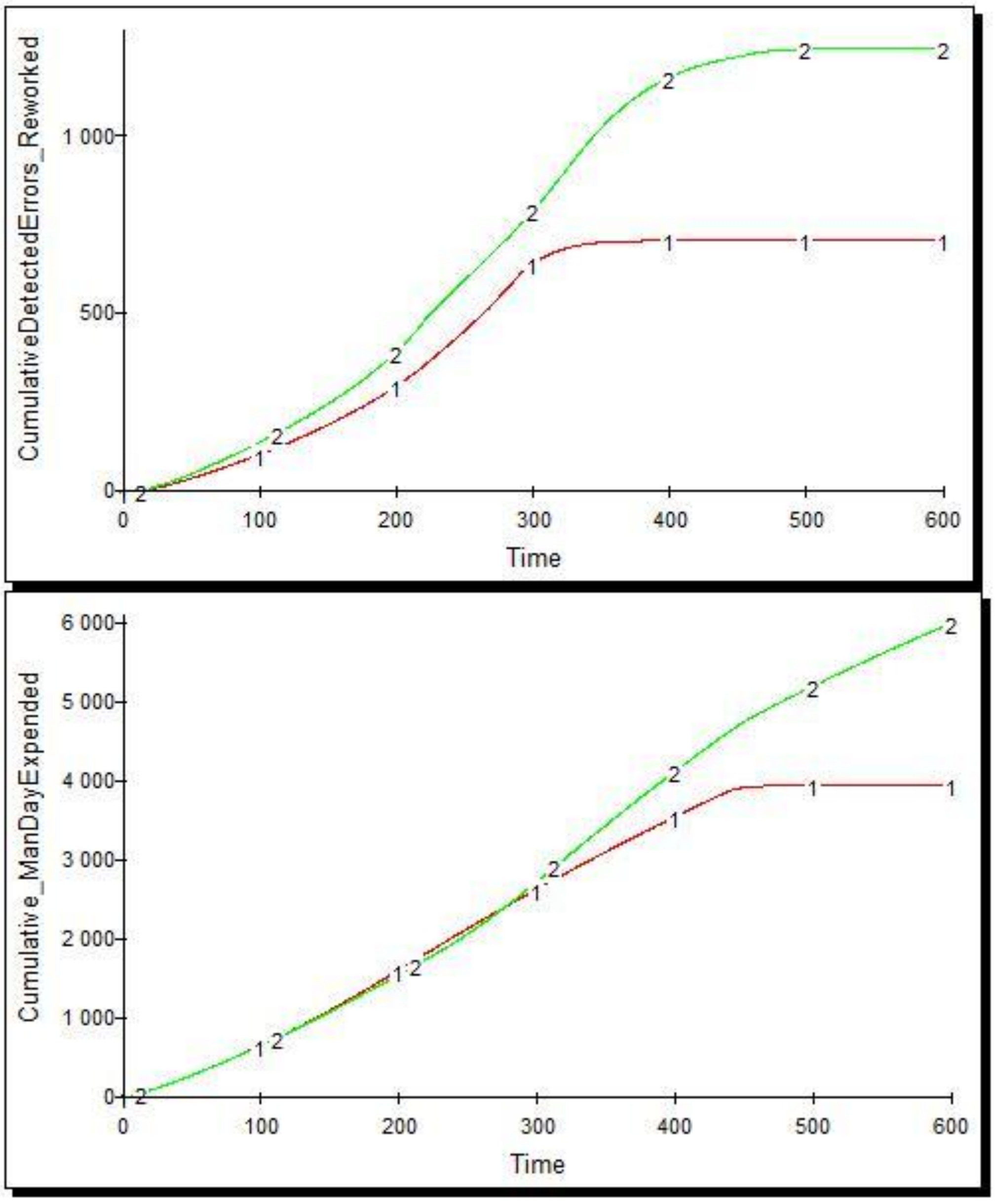


Figure 137. Cumulative detected errors reworked and cumulative man-day expeneded, Red line presents the base run and green line presents the policy run

The next two graphs show the benefit and cost from combined policy 6 compared to the base-run. In my model the cumulative number of detected errors reworked constitutes the benefit, while the cumulated man-day expended constitute cost. For this policy to be an improvement to the original base-run, the detection and rework of errors must outweigh the additional costs of the increase in expended man-days.

From the first graph in figure 137 we observe that the amount of errors detected and reworked quickly increase above the level of the base-run. At first this increase is more modest, as they stem from the effect of a larger quality assurance effort alone. The increase in detection potential leads to an increase in errors detected and reworked, that holds a higher yield than the base-run at all times. The amount of detected errors reworked increases drastically as the testing-regime is activated in the middle of design phase after 20% completion of all tasks. This is in line with the reported results in combined policy 5. Continuous testing is the prime engine for error detection and rework. This effect combined with more quality assurance increases the total amount of detected and reworked errors. In the base-model the total amount of detected and reworked errors level off at 709,40 errors. For combined policy 6 the level reaches a ceiling of 1253.50 detected and reworked errors. This is a bit less than a doubling of detected and reworked errors.

The second graph in figure 137 depicts the usage of man-days. Not surprisingly, the amount of man-days expended increases well above the base-run level. Interestingly, the initial expansion of man-days is not higher when we compare base-run and combined policy 6. In addition, the level of expanded man-days is slightly lower in the middle of the project, that is in the second half of design and in the first half of the coding. This is a depiction of the effect of suspended the quality assurance. When quality assurance is suspended less man-day are expended as a result. However, this has a rubber-band effect. When quality assurance is re-initialized the amount of expended man-days increases beyond the base-run due to the increased need for rework from back-log and additional errors found in testing.

As in the previous analyses, the two graphs in figure 137 displayed for cumulative detected errors waiting for rework and for man-day expenditure contains the characteristic S-shape. The behavior pattern initiates in an exponential growth pattern due to increase in activity and progress in software development. As explained in further detail earlier in this chapter, the potential for error detection and software development increases in the early stages of the software project. This increases both the level of cumulative detected errors reworked, and the man-day expenditure. When the project is closer to completion, the goal of zero remaining tasks is closer. This changes the behavior pattern observed in both graphs, as they now display the goal-seeking behavior pattern. This behavior-pattern is consistent between the base-run and the policy strategies.

In order for this combined policy to be cost-beneficial the overall benefit-cost ratio must be higher than the original base run. The graph is depicted below.

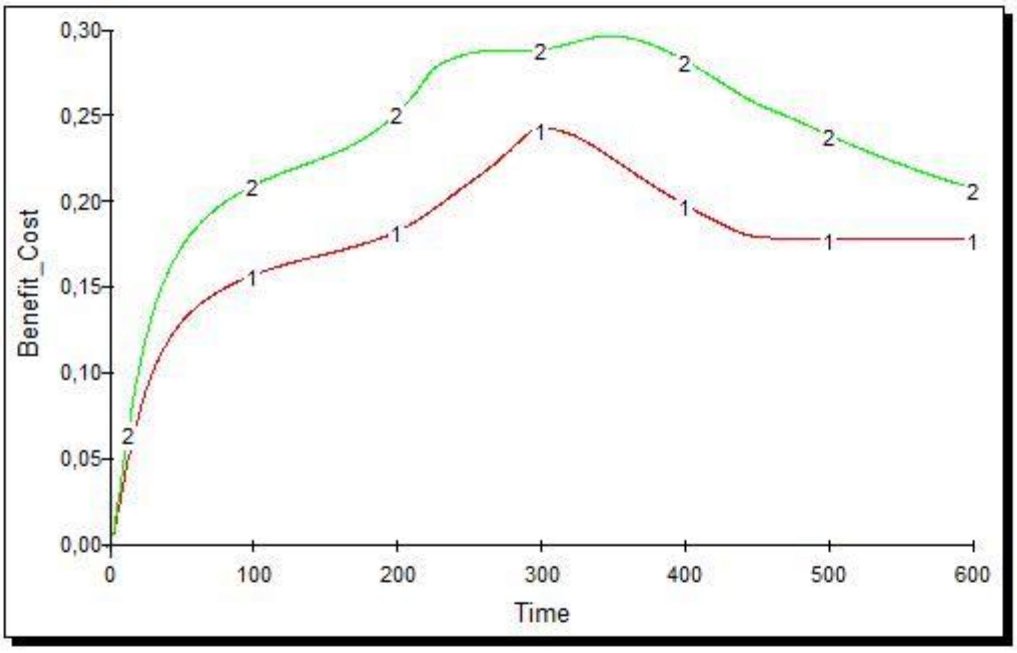


Figure 138. Benefit-cost,
Red line presents the base run and green line presents the policy run

From this graph in figure 138 it is apparent that the combined beneficial effect of continuous testing and an increase in the quality assurance effort outweighs the increase in expanded man-days. We observe that through the entire software project, the benefit to cost ratio is higher than for the base-run. The larger increase in quality assurance allows more errors to be detected and removed from the software. The initial increase provides a better benefit-cost yield than for the base-run, as indicated by the isolated analysis of an increased quality assurance effort.

The policy-option provided by combined policy 6 do indeed yield a higher rate of benefit-cost even though the completion time is later, and the total man-days expended are increased. This follows the argument of Jones and Bonsignour. Their analysis suggests that increased quality of process through quality assurance and testing leads to better quality software. In combined policy 6 a significant amount of errors have been removed from the end-product providing both quality and a better benefit-cost ratio. There is one interesting feature in the benefit-cost graph that should be noted for managerial purposes. The improvement in benefit-cost ratio halts between day 200 to 300. This is the time-period when quality assurance was suspended. The quality assurance suspension did allow more errors to be detected, however, the benefit-cost yield was lower for this time-period. Hence, suspending quality assurance is not a positive long-term decision for a software project.

8.1.3.3 Policy 7: synthesis of policies 3, 1 and 2.

Policy 7 is a synthesis of policies 3, 1 and 2. The policy-strategies investigate here combine the effects of changes in allocation for rework, testing and quality assurance efforts. For the allocation of manpower to systems testing, this is controlled through the variable named "fraction of effort for systems testing". The allocation of manpower for rework is governed by the "desired rework delay". Finally the allocation for manpower to the quality assurance effort derives from the "planned fraction of manpower for quality assurance".

The base-run of this model incorporates the original values for these efforts determined by Abdel-Hamid. Through the "fraction of effort for systems testing" management allocates manpower for testing near the end of the project when 80% of all tasks have been completed. This provides an original scheme of a single system test. This testing-regime will not yield any feedback from detected errors in testing to rework or quality assurance.

The normal managerial strategy for the desired rework delay-time is 15 days. (Abdel-Hamid, 1991: p.75) With this policy it will take 15 days from an error has been detected before it is handled by the rework-effort. This variable is a constant, so for the entire production- cycle 15 days delay is set for all errors detected until they are reworked.

The planned fraction of manpower for quality assurance is determined in the original model to hold a level of 15% of total daily manpower after training. The quality assurance sector provides bureaucratic functions such as walk-troughs, formal interviews and project documentation. In addition the quality assurance effort performs micro-level testing, and identifies and tag errors in the capture-recapture procedure.

This combined policy is a synthesis of the previous explored policy-options in combined policy 5 and policy 6. From the isolated analysis of policy-impact one observed that testing held an important impact on the overall level of error detection and removal. Hence, in this policy-strategy we deploy the continuous testing-regime that initiates earlier in the project, in the middle of design phase of development, when 20% of all tasks have been completed. This strategy will be employed for policy 7 strategy 1 and strategy 2 analysed now.

For manpower allocation to rework the isolated analysis indicated an impact on the ability to rework and correct errors by the desired rework delay-time. From the isolated policy-analysis one, and later confirmed in the combined policy 5, revealed that testing along with both an increase in rework-delay and a decrease in rework-delay yielded a positive result. Hence, for policy 7 strategy 1 the delay-time for rework is set to 10 days. For policy7 strategy 2 the delay-time is set to 20 days.

The final part of this synthesis adds the effect of an increase in the quality-assurance staff. From the isolated policy-analysis two, and later confirmed in combined policy 6, the increase in quality assurance staff is an efficient decision to increase the quality of both process and the final software product. This is the important factor for my analysis, to increase software product quality within a beneficial cost-frame.

This analysis then consists of two policy strategies. Policy7 strategy 1 deploys a continuous testing system after 20% of all tasks have been accomplished. For quality assurance, the planned fraction for quality assurance is set to 20% of the total daily manpower after training. The final element here is the desired rework delay that is set to 10 days. Policy 7 strategy 2 deploys the same continuous testing system as strategy1, and the planned fraction for quality assurance is 20% of total daily manpower after training as well. The difference lies in the desired rework delay, that is set to 20 days. The graphs below in figure 139 depicts the simulations of the base-run designated as a red line with “1”, strategy 1 designated as a green line with “2”, and strategy 3 designated as a blue line with “3” branded on it.

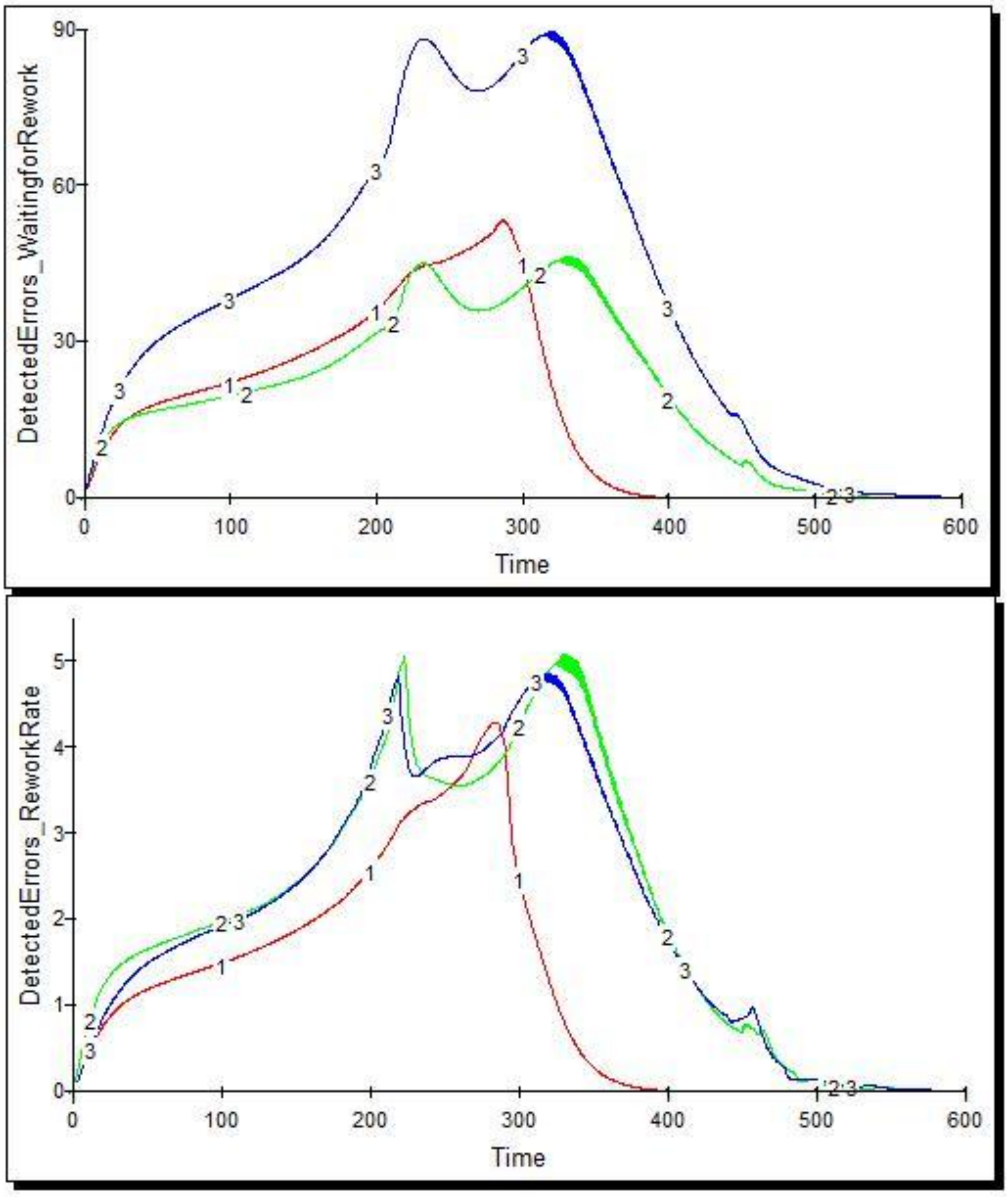


Figure 139. Detected errors waiting for rework and detected errors rework rate,
Red line presents the base run, green line presents the strategy1 and blue line presents strategy2

<u>Policy 7 strategy 1</u>

The software project starts up, and the software development team produces tasks that are sent to quality assurance for error-detection. In this policy-strategy we have a larger quality assurance staff, and more rework-manpower available to correct the detected errors. We observe that initially the increased quality assurance manpower is able to detect more errors. This is due to the potential error detection described by Abdel-Hamid. The potential for error detection is directly influenced by the amount of personnel allocated for the quality assurance effort. This effect combined with a decrease in the desired rework delay, which leads to more allocation of manpower to rework effort, is observable in the two graphs above in figure 139. The amount of detected errors waiting for rework increases at the same rate as the base-run. However, soon the allocation of more staff to rework leads to a lower level of detected errors waiting for rework than in the base run. This is not due to lesser errors detected, but rather an effect of the increased rework rate. In the second graph above we observe how the rework rate holds a significant higher level than the base run. For a period of time the increase in rework rate follows a steep pattern. However, as described earlier in combined policies 5 and 6 the rework effort lowers in intensity due to human factors. The level of fatigue, motivational losses and stress lowers the rework rate. The same effect influences the quality assurance staff, and the level of detected errors waiting rework increases at a slower pace. The behaviour of the graph follows that of the reference mode at this juncture. However, as the testing regime is activated, the change in behaviour becomes apparent.

After 20% of all tasks have been completed, in the middle of the design-phase, the testing effort is activated. This leads to a clear increase in detected errors waiting for rework, and the rework rate. The testing effort identifies tagged and untagged errors that are returned for either quality assurance or rework. The increased quality assurance team is able to identify and send more of also untagged escaped errors discovered by testing team to rework, as well as bad-fixes, as tagged errors discovered in testing are added to the stock through the testing effort. The effect is a sharp increase and peak in the detected errors waiting for rework. The increase in detected errors waiting for rework signals to the management that more effort is needed for rework. In addition, the rework staff has now been able to recover from the initial exhaustion. Due to learning and experience, and the simpler nature of rudimentary design-errors the rework rate increases sharply. This increase is not evident in the base-run, as it is fuelled by the effect of continuous testing. The increase in the base-run follows the increase in software productivity and the number of tasks completed as a result. This effect is also influencing the situation for this strategy 1. More tasks are completed, and as a result the number of errors increases.

As in the previous combined policies, the large amount of errors detected increases the perceived man-days still needed. This leads to an increase in schedule pressure, and a gap between scheduled completion date and actual completion date. In this case, the effect is even more severe. This is due to the fact that the increased allocations for rework, testing and quality assurance manpower have left fewer resources for development. The result is described by Boehm and Abdel-Hamid; suspension of the quality assurance effort, with a reallocation for software production. This occurrence is depicted in the detected errors waiting for rework, and the rework rate. The overall number of detected errors waiting to be rework reduces, at the same time as the quality assurance effort is reduced significantly.

The decline in errors waiting for rework allows management to re-adjust the rework level. In addition to this reduction, also exhaustion, motivational losses and turnovers after a period of high density rework, leads to less rework rate. This should have yielded an increase in detected errors waiting for rework, but from the graph we see an absolute reduction in both variables at once. The reason for this can only be explained by suspension of quality assurance. In this case accomplished tasks that are finished will be sent directly to the testing effort. By suspending the capture-recapture process, all errors picked up by the testing effort are un-tagged errors. This leads to a situation where a back-log of detected errors ends up in the quality assurance department rather than detected errors waiting for rework. Compared to the base-run this behaviour is not following the original model. In the original model, the testing-phase is not active until the very end of the project. The manpower allocated for quality assurance is smaller, and the desired delay for rework is longer. Therefore the sharp increase in schedule pressure leading to a suspended quality assurance effort is not present for the base-run. In fact, the rework rate at this stage drops to a level that is even lower than the base effort for a substantial time-period.

After a period of suspended quality assurance, the effort is reinitialized. The effect on the detected errors waiting for rework, and the rework rate is imminent. The back-log of untagged errors are now identified and sent to rework at a high rate. In addition, the project has now entered the second half of the coding phase. More tasks have been produced, and the production rate is higher. Just as in the base-run towards the end of the project more errors are both found and reworked. However, due to the delay from suspending the quality assurance effort, the peak occurs later here in strategy 1. The suspension of quality assurance also holds a rubber-band effect, as the increase in both detected errors waiting for rework, and the rework-rate returns to the previous peak level. This level is kept until all tasks have been completed by the development staff. From this point on, a larger percentage of the workforce is transferred to the last system test. There are still detected errors that are waiting for correction, and these are corrected at a slowly declining rate. Compared to the base-run this process is done at a slower rate. This is due to the large difference in total detected errors. For this policy7 strategy1 there are more errors to rework and it is delayed due to the suspension of the quality assurance effort. This concludes the analysis for policy 7 strategy 1

Policy 7 strategy 2

The software project initiates, and tasks are being controlled by the quality assurance staff. In strategy 2 the quality assurance staff holds a higher manpower-level than for the base-run. However, the desired rework-delay is set for a longer time-period of 20 days rather than the original 15. The effect of this difference is immediately apparent compared with the detected errors waiting for rework reported in both base-run and strategy 1. The increased quality assurance staff is able to detect more errors, as their error detection potential is high. The rework-effort is slower to initiate in this case, as the desired rework-delay is longer and it leads to less allocation of manpower to rework effort. Therefore we observe that the number of detected errors waiting for rework increases drastically.

When we observe the rework-rate, the increase in detected errors initiates a strong rework-effort. The initial response from the rework-team is high due to positive motivation and a rested work-force. However, due to the size of the rework-team and the longer delay it is not able to maintain as high a rework-level as in strategy 1 with a shorter desired rework-delay and more rework staff in circumstances. Compared with the base-run, the rework effort holds a higher level. This is due to the sense of urgency and schedule pressure derived from a sharp increase in detected errors waiting for rework. This increase leads to a situation where the amount of detected errors signals for management that a focus on rework is necessary. This leads to a cut in slack-time, and the rework-team spends more hours on actual rework as a result. This initial boost in the rework-rate is not maintained for a long period of time, and we observe that the rework rate decreases to a lower level. This is due to exhaustion from the demanding design-errors, motivational losses and fatigue.

The level of detected errors waiting for rework also shows a lower level of increase. This is due to the same factors as explained above for the rework-team. Loss of motivation, fatigue, and communication overheads influences both the development and quality assurance staff.

Design tasks are harder to develop and design errors hold low error density, so error detection requires more attention and a higher effort. Over time this leads to fatigue. In addition, communication between members of the quality assurance team and members of the development staff reduces the productivity of quality assurance. In addition to these explanations of reduction in the detected errors waiting for rework, the allocation of manpower plays a role. With more manpower allocated for quality assurance, less manpower is available for software development. Hence, fewer tasks are completed and fewer errors are generated to get detected by quality assurance.

Halfway into the design-phase, the testing effort is activated. This leads to an increase in the detected errors waiting for rework as well. The testing effort is running applications, and determines whether or not they contain rework-considerable errors. Tasks with such errors are then examined and classified as either tagged or untagged errors. In addition, the effect of learning and relatively simpler design-tasks for production to accomplish provides a higher software production rate. As a result more tasks are accomplished, and more mistakes committed. The combined effect of testing and high error detection potential from quality assurance, increases the stock of detected errors waiting for rework to a high level. This signals again a need for an increase in the rework-effort, and this is observable in the rework rate. Compared with the base-run, this sudden increase in both detection of errors waiting for rework and rework rate, break with the original behavioural pattern. The reason for this is explained through the testing effort. More errors are detected and fed back to quality assurance and rework through the advanced and continuous testing-process that is absent in the original base-run.

The rework-staff has at this point in time been able to recover from the initial exhaustion. With less errors waiting for rework, slack-time was increased and schedule pressures lowered. However, as the testing effort increases significantly the errors waiting for rework the situation changes. Slack-time is cut, and more effort is targeted to the rework effort. This leads to a high rework rate, which for a period of time is able to hold the same intensity-level as strategy 1. However, due to longer delay-time and a subsequently smaller staff this level peaks off at a lower position than for strategy 1.

The initial boost in rework-effort, combined with more manpower allocated for testing and quality assurance provide the same effect on perceived man-days remaining as for strategy 1.

With less manpower available for software development, schedule pressure increases significantly. In addition, the perceived gap between scheduled completion date and actual completion date signals a problem for management. In order to rectify this situation, quality assurance is suspended for a period of time. This is observable in the detected errors waiting for rework stock, as the number of detected errors is reduced. This reduction occurs at the same time as the rework-effort decreases significantly. The exhaustion from the effort combined with motivational losses and turnovers, leaves the rework staff unable to maintain the high level of rework. Interestingly, the behaviour displayed by strategy 2 is different in this situation than strategy 1. Quality assurance is suspended, and more manpower is allocated to the software production effort. However, the total amount of detected errors still waiting for rework signals for management that the effort must be maintained at a high level. The rework rate drops to a level close to the observed level for strategy 1, but increases again more rapidly.

The effect of suspending quality assurance is apparent for strategy 2 as well. The back-log of untagged errors found by testing increases significantly the amount of detected errors waiting for rework. This increase signals to management the need for a final increase in the rework-effort to finish the waiting tasks. The exhaustion-level for the rework-staff has decreased. In addition, the feeling of imminent completion of the rework-effort due to fewer tasks remaining provides a positive motivational factor. The rework-team increases its effort for one last time. This peak is observable in the base-run, but it occurs here in strategy 2 at a later stage due to the suspension of the quality assurance effort. As observed in strategy 1, the rework staff is completing the correction of errors at a slow declining rate. This is due to the slow reduction in staff as they are transferred to the final system test. This behaviour is similar to the base-run but requires more time due to the higher number of detected errors. This concludes the analysis of strategy 2.

<u>Analysis report for completion time, detected errors reworked, man-days expended and benefit-cost ratio</u>

In the first table below I present is the completion-times for the simulations.

Simulation run	Completion Time
Base-Run	438 Days
Policy 7 strategy 1	556 Days
policy 7 strategy 2	548 Days

Figure 140. Project completion time for base run and policy strategies runs

The results in completion times are as expected. For both policy-strategies we were able to detect a larger amount of errors compared with the base-run. In addition, we allocated more resources to the defect-removal effort through continuous testing and quality assurance compared to the base run. For strategy 1 the effect was enlarged by the shorter desired rework delay that leads to more allocation of manpower to rework effort. Hence, software production held a lower level and finished the tasks at a later stage. Between policy-strategies 1 and 2 there is a difference of 8 days. This difference is linked to a relatively higher software production rate due to less manpower allocated to rework by longer desired rework-delay in strategy 2. This difference not a huge difference, but the effect on total amount of reworked errors will be influenced by this element.

The second table that is provided in this analysis reports the total amount of errors that have been detected and reworked through the software production cycle. For my policies to provide better software quality, the total number of errors removed from the software must be significant compared to the original model. The table below contains the total amount of errors detected and removed. This factor constitutes the benefit side of the project.

Policy simulation	Defect removal
Base run	709,40 errors
Policy 7 strategy 1	1263.75
Policy 7 strategy 2	1241.68

Figure 141. Project defect removal for base run and policy strategies runs

The results in table above in figure 141 are in line with the previous reported results from the analysis so far. The changes in allocation for strategy 1 and strategy 2 yield a higher total of corrected errors through the software production cycle. The base run is able to detect and remove 709,40 errors. Compared to strategy 1 and strategy 2 this total is significantly smaller. The inclusion of a continuous testing effort is a significant force for increased error detection. The results from combined policy 7 strengthens the findings regarding testing in policies 3, 5 and 6. In addition the effect of adding a higher manpower-allocation for quality assurance plays a significant role on the total amount of errors reworked. The error detection potential for the effort increases with the quality assurance staff-size. The capture re-capture effort provides a complimentary procedure between testing and quality assurance that detects and identify an even larger rate of errors compared to the base run.

The table provides a second observation concerning the rework effort. From the isolated analysis of policy 1, and the combined analysis of policy 5 we revealed that the desired rework delay did not create a profound effect on the overall defect-removal level. This factor is evident for strategy 1 and strategy 2 as well. The rework-effort is not finding or detecting errors. Rework is performed as an error-correction tool, and therefore the level of errors detected is not sensitive to the allocation of manpower to rework. The ability to rework errors is influenced by other factors such as schedule pressure, loss of motivation and changes in software project standards. This is explained in detail during the isolated policy analysis 1 in this chapter. Hence, the difference between reworked errors between strategy 1 and 2 is not significant compared to the difference between both strategies and the base-run. Therefore, both strategies yield an improvement in defect removal compared to the original model.

The next report concerns the total man-days expended on the project. The total amount of man-days expended provides the total costs for the project. More man-days expended translate to higher costs due to longer production time and a larger amount of total workforce. The table below provides the total man-days expended on the project under strategy 1 and strategy2. These are then comparable to the base-run.

Policy simulation	Man-day expenditure
Base run	3878,70 Man-days
Policy7 Strategy 1	5670.71 Man-days
Policy7 Strategy 2	5614.57 Man-days

Figure 142. Project man-day expenditure for base run and policy strategies runs

From the report on the total man-days spent on the project it is apparent that allocating more manpower to the defect-removal effort causes an increase in the costs of the project as a whole. The first reason for this increase is the fact that manpower is diverted from software development to defect removal. This reduces the potential development rate and the project expends more man-days before completion.

The second reason for this increase is the level of detected errors itself. By detecting more errors at an earlier stage, management must re-adjust both allocation and total manpower-size to cope with the sharp increase in detected errors waiting for rework. As a result, the man-days expended increases due to a longer rework-session and higher total workforce.

The third factor lies in the suspension of quality assurance. As described previously in the analysis, quality assurance was suspended for a period of time due to schedule pressures. Hence, a back-log of untagged errors detected in testing increased the total remaining errors to rework at a late stage in software production. This delayed the completion of the project further. Hence, more man-days where expended on the effort. These elements are evident in both strategy 1 and strategy 2, causing a significant increase in man-day expenditures compared to the base-model. In sum, the increase in software defect removal causes an increase in costs. The remaining question is the relationship between gains and losses through increased error detection versus man-day expenditures.

The next two graphs depict the benefit-side of the policy-strategies through cumulated detected errors reworked, and the cost side through man-days expanded on the project. Unlike the tables, the graphs provide the dynamic development of reworked errors and man-day expenditures through the production cycle. This provides more information on the pattern of development for both benefit and costs in the project as a whole.

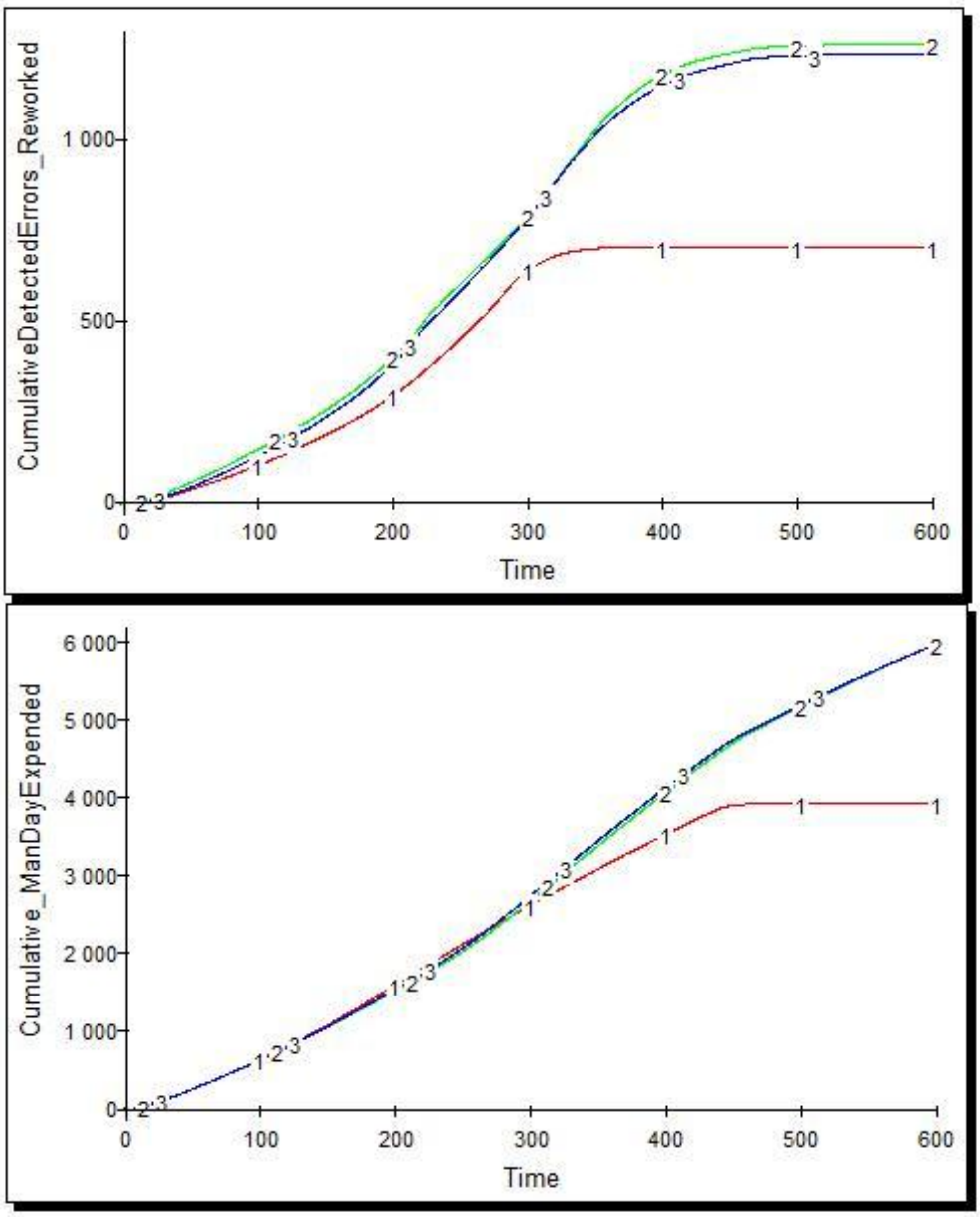


Figure 143. Cumulative detected errors reworked and cumulative man-day expeneded, Red line presents the base run, green line presents the strategy1 and blue line presents strategy2

When we look at the benefit side of the policy strategies, the first graph above in figure 143 shows that more quality assurance and continuous testing leads to a significantly higher level of errors detected and reworked. This indicates a higher rate of end-product quality, and an increase in process quality. Both policy strategies are able to detect and rework a significantly larger number of errors compared to the base-model. Between the two efforts it is apparent that the policy of holding a lower rework delay enables the effort to detect and remove more errors than holding a longer delay. This is in line with the results observed in combined policy 5. The question that now remains is the cost-efficiency of both policy strategy compared with the base run. This is depicted in the final graph below in figure 144 which is the benefit to cost ratio.

When we observe the cumulative man-days expended on the project, both policy-strategies hold a higher level than the base-run. Initially the level of man-days expended follows the same trajectory for base-run, strategy 1 and strategy 2. However, as described by both empirical literature and our previous analyses the continuous testing-effort provides a sharp increase in the man-days expended for the project. As a result, both policy-strategies enter a new increased trajectory of expended man-days that they follow to the end of the project. Interestingly we also observe the effect of the delay by suspending quality assurance. From the initialization of the quality assurance effort, the man-days expended increases significantly. The man-day expending in strategy 1 and strategy 2 follows the same pattern, where strategy 1 reaches a higher level. However, the difference between the two strategies is not significant.

The two graphs above also display the characteristic behavior explained earlier in this chapter. The initial behavior for cumulative detected errors reworked, and cumulative man-day expenditure is an increase as activity in the project initiates. The result is an initial behavior of exponential growth. However, as the project moves towards completion the behavior pattern changes. The goal for the project is zero remaining tasks. When the project progresses towards this goal, the level of detected errors reworked and man-day expenditure levels off towards the goal. The graphs display an S-shape behavior-pattern as a result. This is consistent with the base-run for both strategy 1 and strategy 2 but at a different numerical level.

The last and final graph in figure 144 of my analysis depicts the benefit-cost ratio for policy strategy 1 and 2. For the project to yield a positive reduction in software errors at a cost-efficient level the benefit-cost ratio must be higher for the strategies compared to the base-model.

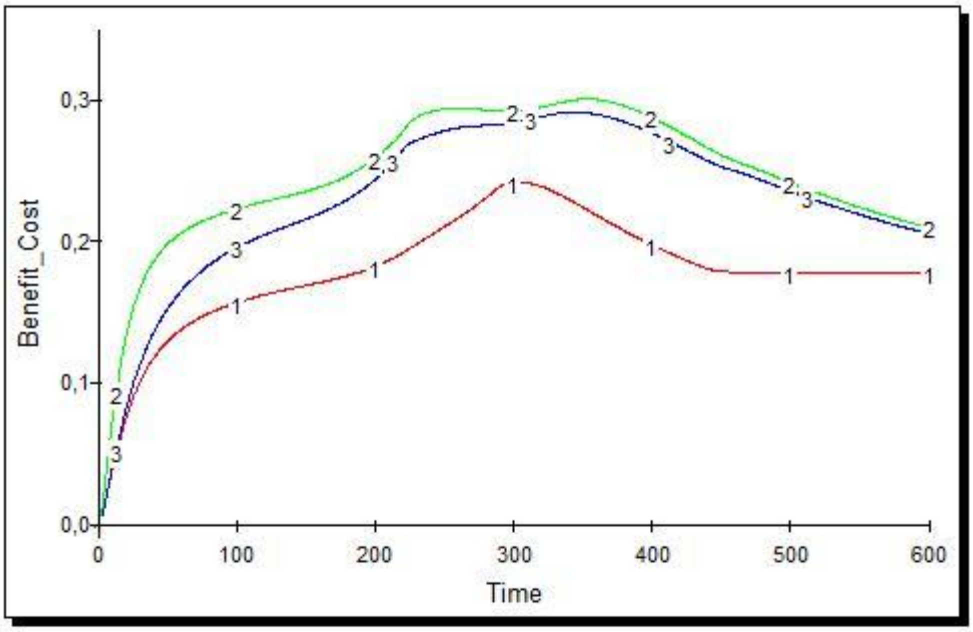


Figure 144. Benefit-cost,
Red line presents the base run, green line presents the strategy1 and blue line presents strategy2

From this graph it is evident that both strategies 1 and 2 yield a higher benefit-cost ratio than the base run. The behaviour of this graph is fairly similar between the base-run and the policy strategies with some notable differences. The additional effort with a larger quality assurance team and shorter rework delay yields a higher benefit-cost ratio for strategy 1 compared to both the base-run and strategy 2. The man-days expended on the additional quality assurance staff is outweighed by the amount of errors detected and reworked. For strategy 2, the initial long delay until the rework effort initiates provide a loss compared to the base-run. More errors are detected but they are not reworked until 20 days have elapsed. Therefore there is not an initial gain from a larger quality assurance effort with a long rework delay.

The significant increase in the benefit-cost ratio in both policy strategies 1 and 2 is the testing-effort. In both graphs we see that the benefit-cost level increases significantly compared to the base-run that is void a continuous testing system. This illustrates yet again the importance of a continuous testing system. The benefit-cost ratio follows the same pattern for the base-run trough the final parts of the software project. More tasks completed, leads to more errors detected and reworked in all of the simulations. It is clear from this graph that the benefit-cost ratio is significantly higher in both policy strategies compared to the base run. This follows the empirical and numerological data provided by Jones and Bonsignour (2012).

<u>**8.1.3.4. Policy 8: Synthesis of policy 3, 1 and 4.**</u>

This combined policy is the first synthesis that contains the client review policy structure. As explained earlier under the isolated analysis of the client review policy, this effort is an extension of the enhanced testing regime. During production, pieces of software are integrated into working applications. These are then tested by the testing effort. After testing, some applications will contain no errors, or minor deviations. Through applying the client review policy these pieces of software are then transferred to the client review stage, in what is known as a client acceptance test. The concept of a client review is established by Jones and Bonsignour (2012), and the effort is able to detect and remove 30% additional errors or deviations from clients need and acceptance from a software product. The inclusion of a client review must be accompanied by a continuous testing effort.

In the original model this process would have been impossible to include, as the testing-effort only contained one final systems test. If one performed a client acceptance test after the final systems test, any feedback or change would have been added post-production. This area is outside the model boundary, as post-production maintenance is not linked to the software production cycle. Hence, any policy-strategy deploying a client review must be based on a continuous testing effort. In addition to this particular characteristic of a client review, it is important to observe the relationship between effort and cost. The client review consists of two separate bodies of staff. One half is designated from the testing-manpower allocated by the software management side. As described earlier in this chapter, the testing staff presents the application for the client. After the client has tested and reviewed the application, the testing staff clarifies significant deviations reported back from the client. The client decides how many people he wishes to allocate and send for the client-review. Hence, the cost of having a client review is not carried by the software project. Most of the actual testing, considering the client review final testing of the applications, is done by the client’s staff, and the bulk of the costs are therefore carried by the client in this case.

In this combined policy, the effects of an enhanced system testing through the allocation of manpower for testing effort changes in manpower allocation for rework effort and applying the client review stage, is analysed. These effects are then compared to the base model run as in the previous synthesised policies analyses presented in this chapter. The original testing regime utilized activates near to the end of the project, after 80% of all tasks have been completed. This allows for only one round of testing and excludes the client review all together. This is governed by the variable "fraction of effort for systems testing." For the allocation of manpower to the rework effort, the desired rework delay is set to 15 days. This entails a period of 15 days between an error is detected, until it gets reworked.

For the combined policy we wish to analyse the effect of manpower allocation for rework, together with an enhanced testing regime and applying a client review stage. For policy 8 strategy 1 we set the testing effort to commence earlier in the project, in the middle of the design phase of development, after 20% of all tasks have been completed. This provides the crucial continuous testing needed for the inclusion of a client review stage. In regards to the manpower allocated for rework, the desired rework delay is set to 10 days rather than the original 15, which causes more allocation of manpower for rework effort. For policy 8 strategy 2 we investigate the other allocation-strategy for the rework process. In this case, the desired rework delay is set to 20 days rather than the original 15 day, which causes less allocation of manpower for rework effort. The testing effort follows the same allocation as for strategy 1. The results are depicted in the two graphs below in figure 145. Just as in the previous analyses, the base run is the red line with a "1" brand. Policy 8 strategy 1 is the green lined with the "2" brand, while the policy 8 strategy 2 is the blue line with the "3" brand.

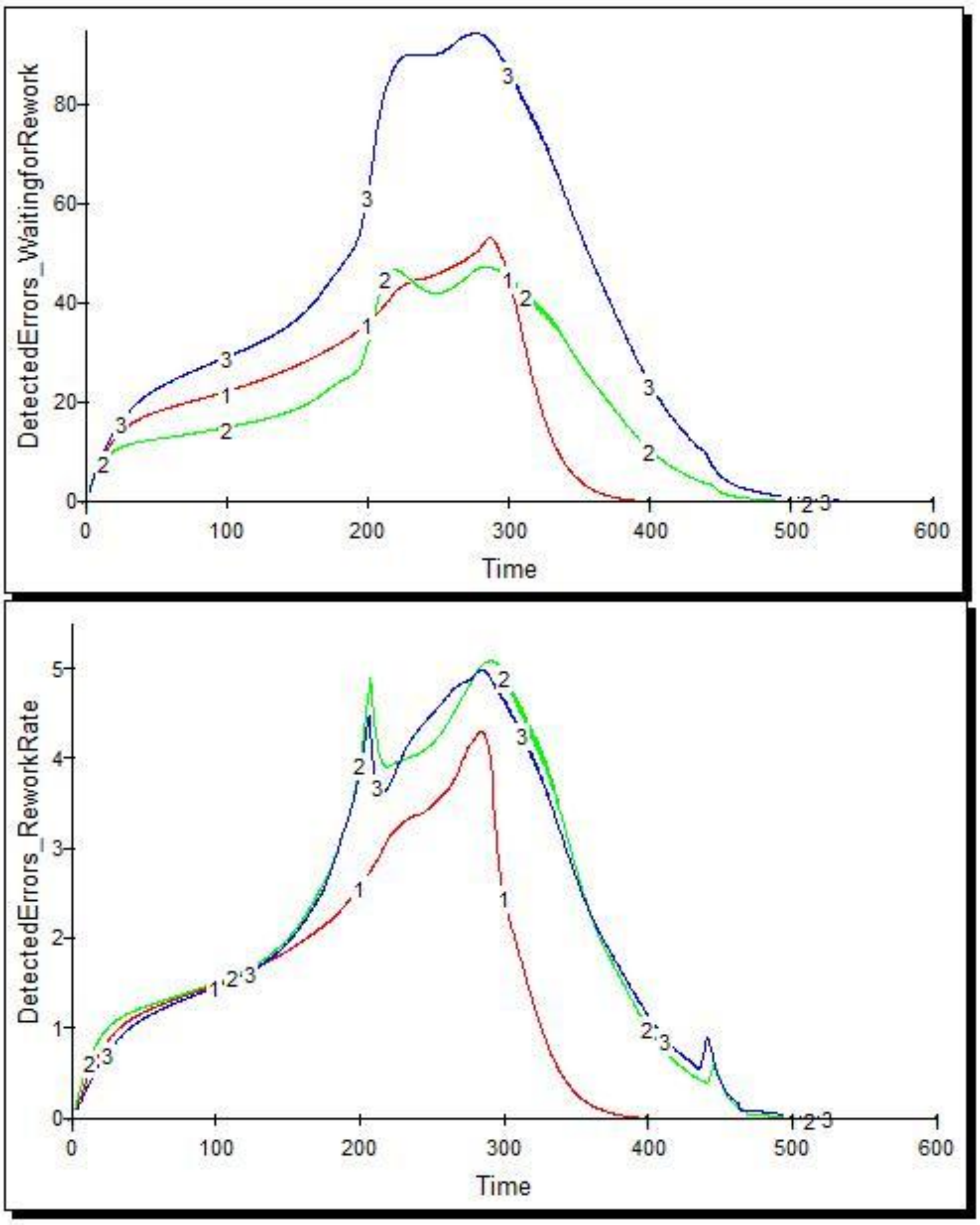


Figure 145. Detected errors waiting for rework and detected errors rework rate, Red line presents the base run, green line presents the strategy1 and blue line presents strategy2

<u>**Policy 8 strategy 1**</u>

The software project starts and the number of completed tasks are sent to the quality assurance effort. They detect errors and the stock of detected errors waiting for rework increases accordingly. As shown it is in Figure 145, the graph of detected errors waiting for rework, the effect of increased allocation for rework effort through the reduced desired rework delay is apparent at the first stage. The rework team starts to correct the detected errors at a high rate, which is observable in the detected error rework rate. As explained in detail in combined policy 5, the initial rework effort is higher, and the amount of detected errors waiting for rework holds a lower level through the first period compared to the original base-run. The amounts of detected errors continue to increase steadily through the initial stages of the project but at a significantly lower level than compared with the base run.

The reason for this increase rests with the rework staffs ability to perform correction, and the number of tasks completed. Initially the rework team starts the process rested, and eager to complete the correction-tasks. This is a positive motivational factor, which is beneficial for the process. This is observable in the rework rate which starts out with an initial strong increase in the rework rate. However, this rate of rework is impossible to maintain indefinitely. After a period of time the rework team starts to feel fatigue from the initial boost. Reworking design errors demand a high level of effort due to the fact that the errors are fundamental in the software fabric. Hence, the rework rate starts to decrease slightly. The second reason for the increase in detected errors waiting for rework lies in productivity. When the project starts, the first tasks completed are fundamental design-tasks containing complex codes. As the project processes, the software teams move into more rudimentary design tasks. The result is an increase in completed tasks sent for quality assurance. More tasks accomplished translate into more errors being committed and the amount of detected errors increase steadily.

Midway through the design phase, nearly 20% of all software tasks have been completed. This initiates the testing-effort. The enhanced testing regime performs functional tests on applications in order to determine their suitability. As explained earlier, the testing effort identifies the rework-considerable errors as either tagged or untagged errors. These errors are now returned to either the quality assurance effort or the rework effort. This process provides a sharp increase in the total amount of detected errors waiting for rework.

The response to this increase in detected errors waiting for rework is visible in the rework rate. The rework team has been able to recover from the initial impact of the rework effort and the rework rate increases rapidly. Compared with the base-run this increase is significant. Secondly, the project is now situated in the final parts of the design phase. Errors from this phase are simpler to correct, and the rework team has been able to gain valuable knowledge of the inner workings of the software project through learning during production. With an increase in detected errors waiting for rework, the sense of urgency and schedule pressure cuts slack time, and the rework team puts more actual hours into the correction effort. In addition, this increase in detected errors waiting for rework signals to management that the total staff-level should be adjusted. Management re-adjusts the workforce upwards in response to this development.

Again the response in rework rate is increased, and reaches a peak as the project moves from design to coding. Coding errors are fundamentally easier to rework, and therefore the effort needed per average error decreases. The boost in the rework-rate decreases the amount of detected errors waiting for rework. The previous increase in the rework rate cut of slack-time and the rework team has reached its exhaustion level. The rework effort returns to a lower rate, visible in the sharp rework-rate decline after the peak. This behavior is not observed in the base-model due to the lack of a continuous testing system as described earlier in combined policy 5.

At this point in time we observe a second change in behavior for the stock of detected errors waiting for rework and the rework rate. This is occurring due to the influence of applying the client-review stage. If we compare the graphs in figure 145 with the graphs in combined policy 5, there is a distinct new and significant second peak in detected errors waiting for rework that is slighter and lower in combined policy 5. The difference between these two policy-analyses is the client review stage. By applying the client review stage, the applications that have been deemed no-rework considerable after system testing are transferred for the client review. Here the client's staff has been given the opportunity to review the software and to provide their opinion of its functionality. The client review is able to remove an additional average of 30% of errors and deviations from client's acceptance and needs from the software according to Jones and Bonsignour (Jones and Bonsignour, 2012). This translates to yet another increase in detected errors waiting for rework that are fed back from client review stage. Deviations that have been deemed important for rework by the client is now transferred back to rework.

This causes a new, more significant and higher second peak in the rework rate to accommodate this new level of detected errors waiting for rework. The rework staff is able to muster one last push as imminent completion provides positive motivation. This increase in detected errors waiting for rework signals to management that the total staff-level should be adjusted. Management re-adjusts the rework workforce upwards in response to this development. Additionally the rework-team has been able to recover from the initial increase in effort.

This peak occurs at a later stage after the first peak and not as one continuous increase. This difference is due to the process and information delay in the client review stage. The feedback from the client review stage is delayed due to the presentation and post-review analysis by the testing-staff. The errors detected in this process are fed back during the continuous system effort, and after the effort has halted as all development tasks have been completed. Secondly the rework staff recovers from the previous exhaustion and is motivated to increase their effort again. Therefore they cut-slack time and increase their rework rate. The Final reason rests with the fact that a higher production rate gives more finished tasks and more detected errors to correct. In addition the continuous testing effort and the client review provide more errors that need to be corrected.

This increase in rework rate is maintained until all tasks have been completed by the software development team. As explained earlier, the manpower is transferred from development, quality assurance and rework to the final system testing. There are still errors that need to be reworked, and the rework effort continues until they are all corrected. Due to the higher level of detected errors, the effort needs more time to finish. The graph shows the incremental decline in the rework rate as more staff is slowly moved to the final systems test. This concludes the analysis of policy8 strategy 1.

Policy 8 strategy 2

In the policy 8 strategy 2 the desired delay for rework is set to 20 days rather than the original 15 days. This is done to analyse the effect of applying the client review stage on a policy that contains the continuous testing stages and lower allocation of manpower for rework effort through a longer desired rework delay. As in strategy 1, the project is initialized and software is produced. Tasks are completed and sent to quality assurance, where errors are detected. While tasks are developed, errors are generated as well. The effect of the longer desired rework delay is clear from the initial stages of the software project. This decision causes the rework effort to start later, and less manpower is allocated to this effort. The result is a sharp increase in detected errors waiting for rework. Quality assurance is detecting errors and sending them for rework, but the smaller staff is not able to rework them at the rate observed in both the original base-run and strategy 1. In addition, a smaller rework-staff translates to more allocated of manpower to development effort and a larger production staff. This equals more tasks accomplished that contain more total amounts of errors.

In addition to the factors mentioned above, the negative effect of exhaustion, turn-over, schedule pressure and loss of motivation impacts the rework team. Due to positive motivation, and a rested crew the initial rework rate increases sharply. However, due to the longer delay time and smaller rework staff this initial effort is lower than the base-run. The rework team is also unable to maintain the same amount of rework efficiency for a long time-period. The effects of exhaustion due to the complexity of design errors level causes the rework rate to level off in the same manner as for the base-run and for strategy 2. As a result, there is a larger increase in detected errors waiting for rework through the initial stages of the project. This is clear from the amount of the stock of detected errors waiting for rework. Compared to the base-run and strategy 1 the amount of detected errors waiting for rework increases significantly.

After the first half of the design phase of development has passed, nearly 20% of all tasks have been completed. This activates the testing effort. Just as observed in previous analyses, this effort increases the detected errors fundamentally. The behavior pattern observed in both the stock of detected errors waiting for rework, and the detected errors rework rate is significantly different compared to the base-run.

The sudden increase in detected errors waiting for rework causes management to re-adjust the total manpower upwards. The increase in schedule pressure, and the sense of urgency motivates the rework team to cut slack-time and correct more errors. In addition, the final half of the design phase contain errors that are relatively simpler to correct compared to fundamental design tasks. The rework team has also gained experience, and learned more about the inner workings of the software under production. These factors all contribute to a peak in the rework rate. This peak provides for a short period of time an increase in the rework-effort that equals the rate of new detected errors. For a smaller rework-staff this effort is not maintainable for a long period of time, and the team is exhausted. Schedule pressure, loss of motivation, and turnovers decreases the rework-rate to a lower level.

At this juncture in time, feedback from the client review stage is entered. The deviations found by the client review are now clarified and sent back to rework stage as detected errors to get rework. This constitutes the second increase in detected errors waiting for rework observable in the stock of detected errors waiting for rework. Compared to the graph for combined policy 5, this development is new, higher and more significant, and can only be attributed to the client review effort. The increase in detected errors signals again for management that manpower must be adjusted for completing the newly detected errors. Again the rework-rate climbs into a higher rate, and it even surpasses the original peak at the end of the design-phase. More manpower is allocated and assigned for rework and errors are corrected at a high rate. This effort benefits from an increase in manpower and positive motivation as the rework effort is close to completion.

Just as in policy strategy 1 the production tasks have been completed, and staff is moved to the final system testing. Due to the high number of detected errors, this effort needs more time to be completed. Therefore the detected errors waiting for rework and the rework rate decreases in a smooth fashion towards zero. This effort requires more time than in the base run, which is evident by comparing the red and blue lines. This concludes the analysis of strategy 2.

<u>**Analysis report for completion time, detected errors reworked, man-days expended and benefit-cost ratio**</u>

Just as in the previous analyses, the first table provides the total amount of days spent to complete the software project.

Simulation run	**Completion Time**
Base-Run	438 Days
Policy 8 strategy 1	501.5 Days
Policy 8 strategy 2	501 Days

Figure 146. Project completion time for base run and policy strategies runs

The results from the addition of a client review reveal the same observed effect as in policy 4. The base run finishes at the reported 438 days. The inclusion of a continuous testing system and the increase in detected errors leads to a longer completion time for combined policy 8 strategy 1 and strategy 2. Since more manpower was distributed to the defect-removal effort in both cases, it is expected to have a longer production time as well. More manpower for defect removal translates to less manpower available for the software production effort. This pattern is consistent with the reported completion time in combined policy 5. However, when we observe the completion time the reported completion for both policies is shorter than for combined policy 5. Due to this result, ccombined policy 8 reveals an interesting pattern for the completion time in my analyses.

From policy 4 we learned that including the client-review stage provided better information on the true amount of rework needed to complete the project. This factor increases management's perception and accuracy for allocation to rework. In addition, the increased level of detected errors deriving from the client review at a later stage increases management's workforce-level to accommodate this development. Manpower is transferred from quality assurance and rework to the final system testing as the project has finished all tasks. However, since manpower is allocated to the testing-effort after 20% of the tasks are developed, the initial rate of transfer from quality assurance and rework after completion needed is less compared to the base-run. Hence, more personnel can be kept for the rework-effort in order to finish the remaining correctional activities. This is evident by comparing the combined policy 8 with combined policy 5.

The difference between the two syntheses is the client review. This phenomenon was explained under policy 4 in the following manner. As the client review is a continuous effort in the same way as the enhanced testing effort. This provides management with better information to the perceived time still needed to complete the project. The available information on errors provided by the enhanced testing effort and the client review allows management's perception of rework-rate needed to be more accurate. Secondly the addition of errors found by the client review towards the end of the project, lead management to adjust the rework-staff upwards to accommodate the increase in errors. As a result the total amount of manpower allocated to rework at the end of the project is higher. When manpower is transferred to testing, the continuous system has allocated workers to this effort previously after 20% of all tasks where completed. Hence, the need for manpower to the effort is lower than in the base-run.

When manpower is transferred between development, quality assurance and rework to testing, the perceived level of rework still needed change the distribution of manpower-transfer. More rework-personnel are therefore available for the effort to finish the last correction tasks, and the project finishes at an earlier point in time. This factor establishes the beneficial information-side of the client review effort. Management is able to allocate more accurately to the rework effort when more information on the actual need is available. From this explanation and the results from combined policy 8, I predict that the same pattern will be apparent in the following combined policies incorporating the client review stage. This pattern will be investigated after the completion of the analyses.

The next step in the analysis is to investigate the reported total of detected and reworked errors for each of the policy strategies. For this analysis the effect of continuous testing, desired rework delay and the client review stage is under investigation. The amount of errors removed from the software constitutes the benefit side for my analysis. Hence, the ability to remove errors is an important factor. The table below contains these results.

Policy simulation	**Defect removal**
Base run	709,40 errors
Policy 8 strategy 1	1107.06 errors
Policy 8 strategy 2	1090.19 errors

Figure 147. Project defect removal for base run and policy strategies runs

For this combined policy, the strategies are improving the total amount of errors removed from the end-product. In the base model, the defect-removal effort was able to remove 709,40 errors. This amount is significantly smaller than the reported defect-removal result for strategy 1 and strategy 2. Between the two strategies, the difference is 16,87 in favor of policy strategy 1. This is not a large difference, but there is an increased improvement for policy strategy 1 none the less.

The reason for this relatively close defect-removal total lies in the nature of rework. The prime engines for defect removal in combined policy 8 are the testing and client review effort.

The rework effort is not detecting errors, but correcting them. The increase in error detection for combined policy 8 rests therefore with the other defect removal efforts. As explained in combined policy 7 , the isolated analysis of policy 1, and the combined analysis of policy 5 revealed that the rework delay-time did not create a profound effect on the overall defect-removal level. Therefore the difference in defect-removal between strategy 1 and strategy 2 mimics these results. The ability to rework errors is influenced by other factors such as schedule pressure, loss of motivation and changes in software project standards. I explained this element in detail during isolated policy analysis 1 in this chapter. Hence, the difference between reworked errors between strategy 1 and strategy 2 is not significant compared to the difference between both strategies and the base-run. Therefore, both strategies yield an improvement in defect removal compared to the original model.

The third and final table contains the man-days expended on the project. As described earlier, the costs of software projects are measured in the man-days expended during the software development cycle. The table below reports the total amount of man-days expended by the base-model together with policy strategies 1 and 2.

Policy simulation	**Man-day expenditure**
Base run	3878,70 Man-days
Policy 8 Strategy 1	5469.59 Man-days
Policy 8 Strategy 2	5385.12 Man-days

Figure 148. Project man-day expenditure for base run and policy strategies runs

From the report on the total man-days spent on the project it is apparent that allocating more manpower to the defect-removal effort causes an increase in the costs of the project as a whole. The first reason for this increase is the fact that manpower is diverted from software development to defect removal. This reduces the potential development rate and the project expends more man-days before completion.

The second reason for this increase is the level of detected errors itself. By detecting more errors at an earlier stage through the continuous testing effort, management must re-adjust both allocation and total manpower-size to cope with the sharp increase in detected errors waiting for rework. The client review stage strengthens the effects of the factors highlighted above. Due to the client review stage, the process and information delay causes a second peak of increase in the stock of detected errors waiting for rework. This causes again a longer rework-session to correct and remove all detected errors at the end of the production cycle. Management is forced to allocate yet again more manpower to the rework-effort, and as a result, the man-days expended increases further. These elements are evident in both strategy 1 and strategy 2, causing a significant increase in man-day expenditures compared to the base-model. In sum, the increase in software defect removal causes an increase in costs. The remaining question is the benefit to cost ratio that determines the difference between gains and losses through the increased error detection effort versus man-day expenditures.

The benefit and cost development is depicted on the next two graphs in figure 149. The benefit for the policies is measured by the cumulative detected errors reworked. The more errors that have been detected and reworked translate to a higher degree of software quality and process quality. On the cost side I report the cumulative man-days expended on the project. In order to calculate the benefit-cost ratio these two elements must be taken into consideration.

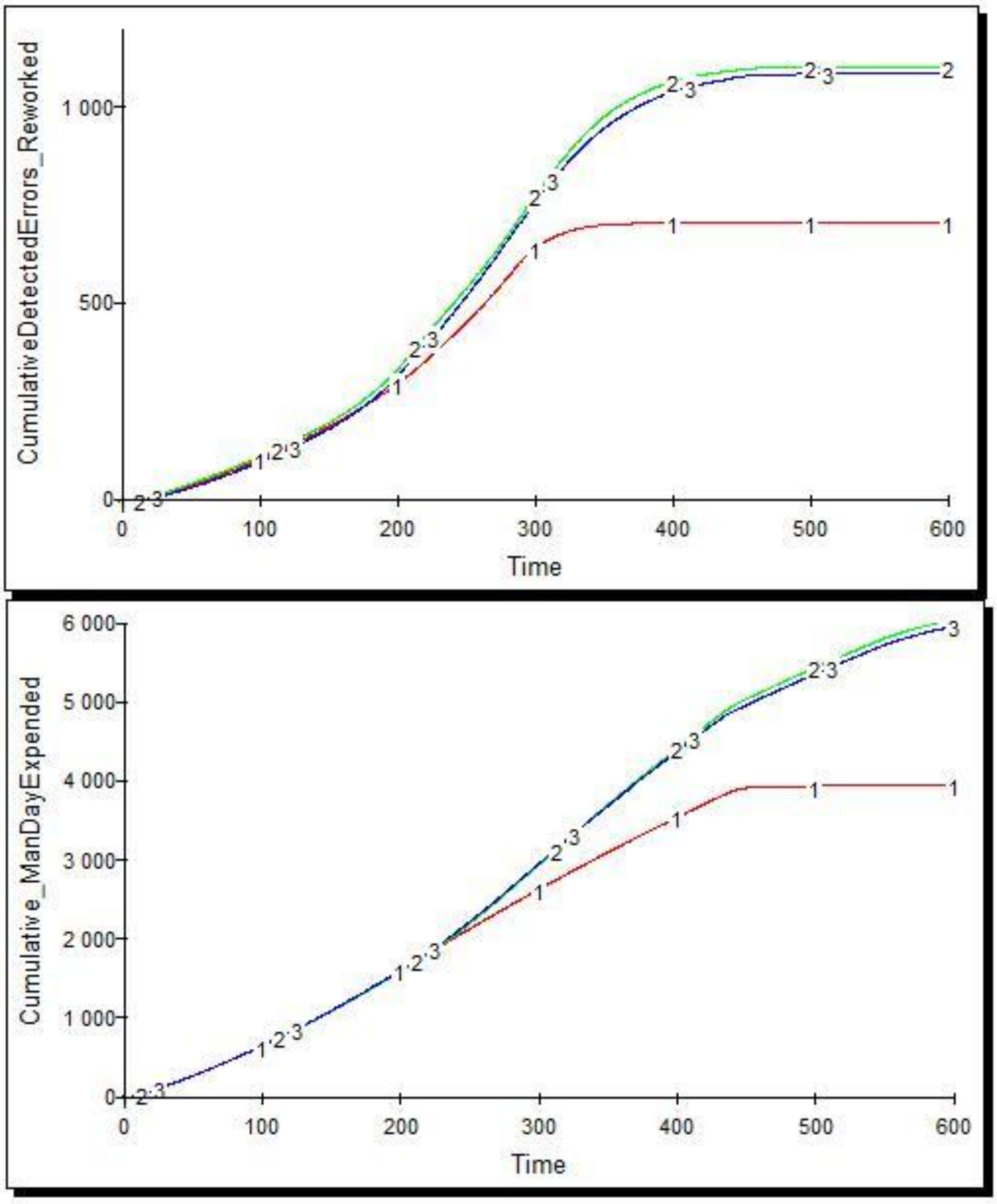


Figure 149. Cumulative detected errors reworked and cumulative man-day expeneded, Red line presents the base run, green line presents the strategy1 and blue line presents strategy2

The first graph above in figure 149 represents the cumulative detected errors reworked by both strategy 1 and strategy 2 as the benefit of the project. This rate is significantly higher than the base-run. The base model before applying the policy is able to remove 709.40 errors. With an enhanced testing regime and a client review, the total amount of errors detected and removed is nearly more than 1.5 times more than the original amount. This is a significant increase that establishes the importance of both testing and client review stages for the defect removal effort.

The second graph in figure 149 represents the cumulative man-days expended in the project by both strategy 1 and strategy as the cost of the project. The cumulative man-days expended on the project are higher than for the base-run. This is as expected, taken the completion-time into consideration. The testing effort and the client review stage detected more errors than the base-run. To accommodate this higher level of correction, more man-days have to be expended. This is clearly depicted in the second graph above. In sum, the cumulative man-days expended for both efforts are significantly higher than for the base-run.

The behavior observed in the two graphs above depicts the characteristic behavioral pattern reported in the analysis of my enhanced model. The initial development follows that of exponential growth for both base-run and policy strategy. However, the difference in allocation changes the level for the policy strategy compared to the base-run. As the project progresses the end, it reaches its goal of zero tasks remaining. At this point the increase in man-days expended and detected errors waiting for rework levels off towards the goal. This provides the S-shaped pattern explained previously in this chapter.

The final analysis is the benefit-cost ratio. For the policies to improve the quality of the end-product within a cost-beneficial frame, the increase in errors detected and removed from the software must outweigh the increase in expended man-days. This is analysed in the graph below in figure 150.

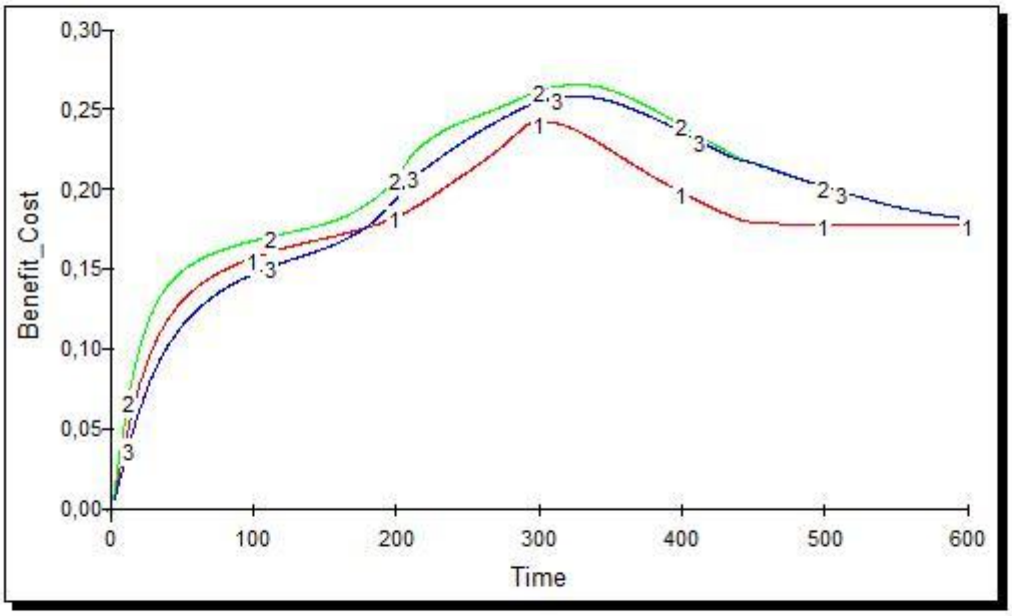


Figure 150. Benefit-cost ratio,
Red line presents the base run, green line presents the strategy1 and blue line presents strategy2

From this graph it is evident that both strategies 1 and 2 yield a higher benefit-cost ratio than the base run. The pattern of this graph is similar between the base-run and the policy strategies with some important observations. The additional rework effort due to a shorter desired delay for the rework effort yields a higher benefit-cost ratio for strategy 1 compared to both the base-run and strategy 2. For strategy 2 the decision to have a longer delay until the rework effort initiates causes a loss compared to the base-run. More errors are detected but they are not reworked until 20 days have elapsed. Therefore there is no an initial gain compared to the base run until the testing effort is initiated.

The significant increase in the benefit-cost ratio for both strategies 1 and 2 is again the testing-effort. In both graphs we see that the benefit-cost level increases significantly compared to the one-test strategy of the base-run. This illustrates yet again the importance of a continuous testing system. In addition we observe that the inclusion of a client review has yielded more errors detected and reworked without causing a loss in the benefit cost ratio. Since the main cost of a client review, that is that amount of man-days spent by the client, is covered by the client himself, the additional errors are detected without causing more man-days to be expended. There is a clear benefit in providing the client with the opportunity to test and review the software before it is completed. Better end-product quality follows as a result. This concludes the analysis of combined policy 8.

8.1.3.5 Combined policy 9: Synthesis of policies 3, 2 and 4.

The 9th policy in this chapter is the combined policy of enhanced testing, the client review stage and the allocation of manpower to the quality assurance effort. This policy combines the previously analysed effect of having a continuous testing system with an increase in allocation of manpower for quality assurance effort, together with applying the client review stage. From the isolated policy analysis of manpower allocation for quality assurance, the isolated policy analysis of manpower allocation for testing effort and the previous analysis of combined policy 6, that is the combined effect of these two isolated policies, we revealed the significant increase in error-detection and removal having a continuous testing regime and a higher manpower allocation. This effect will now be analysed together with the effect of introducing and applying the client review stage to the software production cycle.

The base-model contains the same formula as for the other policy-analyses in this chapter. “The fraction of effort for systems testing” governs the allocation for the testing phase. In the original model the effort starts near to the end of the project, when 80% of all tasks have been completed. This design provides a single systems test. Through the “planned fraction of manpower for quality assurance” the allocation for manpower is set to 15% of total daily manpower after training.

For policy strategy 9, the enhanced testing-regime is deployed. This design initiates the testing effort earlier in the project, in the middle of design phase of development, after 20% of all tasks have been completed. The testing-effort is continuous through the software production cycle. The effect has been well documented through all the previous combined analyses, and the isolated analysis for allocation of manpower for the testing effort. In addition to this testing-regime, the client review stage is activated for this combined policy. In regards to the allocation of manpower to the quality assurance effort, the fraction allocated for this effort is set to 20% of the total daily manpower after training. This policy will now be analysed and compared with the base-run in order to determine the effect of client-review on a policy with more manpower allocated to the QA effort. The results are presented in the graphs below in figure 151.

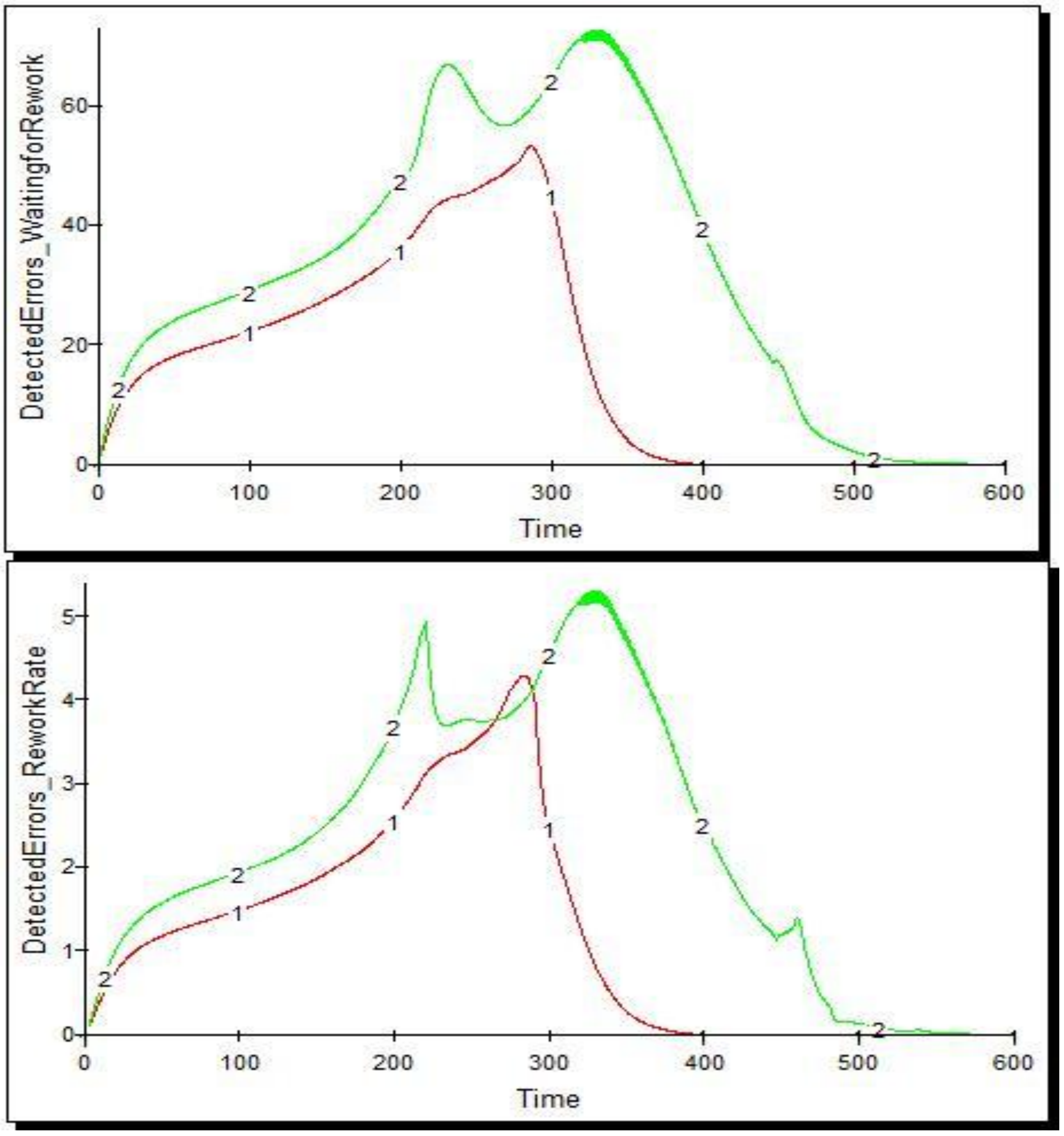


Figure 151. Detected errors waiting for rework and detected errors rework rate, Red line presents the base run and green line presents the policy run

As in combined policy 6 we observe that having an increased quality assurance staff, provide an immediate impact on the errors detected at the early stage of software production. When the software project starts, the main tasks constructed and completed are design errors. This provides a low error density, and it takes more effort for quality assurance to detect and find these errors. In the stock of detected errors there is a sharp increase in detected errors waiting for rework, as the larger quality assurance team detects errors at a higher rate. This increase in detection holds a significant influence on the rework rate as well. The initial rework-rate holds a strong increase at the first stages of the rework-effort. There are two factors playing in at this point in time. The initial rework rate is high due to a rested and eager rework-staff, and that positive motivational factors are present. In addition, the early increase in error detection sends a signal for management that the rework effort must initiate with a higher efficiency due to more detected errors. Compared to the base run, the increase in the quality assurance staff provides a higher degree of detected errors and a higher rework rate than the original model did.

As observed in earlier analyses, the rework staff is unable to maintain the rework rate over time. The negative impacts on the rework rate such as fatigue, stress and loss of motivation decreases the rework rate for a period of time, and the rate holds a slower but steady increase. The rework rate maintains a higher level than the original base-run for a long period of time in the software project cycle. This is due to the increase in the potential error detection that is governed by the size of the quality assurance staff (Abdel-Hamid, 1991: p). The detected errors waiting for rework also decline in increase as the observed rework rate. The reasons for this lower increase in detected errors waiting for rework are similar to those factors that influence the rework rate. The development and quality assurance staff is susceptible to the same motivational factors. This is described in detail in combined policy 6. Fatigue, motivational losses and communication losses all decrease the development and quality assurance effort's productivity. The result is a lower increase in detected errors waiting for rework.

The second effect in play is the allocation of resources and number of tasks completed. Since the increase in detected errors occurs at a high rate, management allocate more manpower for the rework effort as a response. Hence for policy strategy 9 more personnel have been transferred to the quality assurance and rework efforts, moving potential resources from development to defect detection and removal. Quality assurance teams are only concerned with the bureaucratic functions of production, defect detection, and capture re-capture. Less manpower available for development leads to fewer tasks produced. With fewer tasks to investigate, fewer errors are found. The third influencing factor is learning and experience. Since the project is moving from fundamental design to more rudimentary design, the production team has increased knowledge of the software they are creating. This allows the production team to accomplish more tasks with fewer errors.

After a time-period with this relative smooth increase in detected errors waiting for rework, the effect of testing enters into play. It happens in the middle of the design phase of development, when 20% of all tasks now have been completed, and the errors detected are fed back to either quality assurance or rework. This facilitates another sharp increase in detected errors waiting for rework, and we observe a continuous increase that results in a peak near to the end of design phase of development.

In addition to the effects of testing, the quality assurance staff has been able to recover from the initial exhaustion. In addition, the quality assurance staff and production staff have been able to incorporate a more effective communication-level. Experience, personal connections between staff-members, and predictable routines allows for smoother communication and fewer time-losses for communication. This increases the quality assurance staff's error detection potential, and more errors are being detected. In this time-period the project has moved from design to coding, allowing more tasks to be completed at a higher rate from the development-side. More tasks completed leads to more errors being committed. Fundamental coding tasks still require more effort than the final and simpler coding tasks, so errors do occur at a higher rate. Since the rate is higher, the error density increases as well. Higher error density increases the quality assurance staff's potential for detection. Higher error density reveals error-patterns that reveal chains of errors.

From the rework side we observe the same increase in rework rate. The initial exhaustion of the rework staff has diminished, and they are able to rework more errors. Since these errors are more coding-errors and simpler to rework, more errors are reworked by the same amount of rework-staff. With the increase in detected errors waiting for rework supplied by the testing-effort and the increased quality assurance detection potential, management is given a signal for an increased rework-effort. The amount of detected errors waiting for rework increases in such a manner that schedule pressure increases. Slack-time is being cut, and more effort is targeting the defect removal effort. This is observable in the rework rate, as the effort increases rapidly and peaks at the end of the design-phase.

Compared with the base-model the effect of the testing effort is clearly visible in both graphs. The increase in both the detected errors waiting for rework, and the rework rate climbs with a steeper trajectory. The increase is significant, and is caused by the testing effort that is initiated at this point in time. The base-model also displays a peak in the rework and detected errors waiting for rework rate at this time. This peak is not nearly as profound as the development in combined policy 9, both in level and trajectory.

After this initial increase and high peak in detected errors waiting for rework, we observe a change in behaviour that is unlike that of the base-run as explained in the analysis of combined policy 6. For both the stock of detected errors waiting for rework, and the detected errors rework rate there is a drastic decrease. Initially from the stock of detected errors it looks like the increased rework effort is able to rework and remove more errors than quality assurance and testing is able to find more errors. However, when we look at the rework rate its apparent that this is not the case. Both stock and rate is decreasing at the same time. The reason for this behaviour lies in the policies we are applying here, and the production of software.

In this policy the total manpower that is assigned to the defect-removal effort is larger than in the original base-run. Due to a higher amount of detected errors and the necessity for a high rework-rate, less manpower is available for the software development itself. In policy 6 the effect from the increase in schedule pressure with the increase in perceived task still remaining, held a significant effect on the quality assurance effort. For policy strategy 9 the same effect is entering into play. Management observes that there is a significant gap between scheduled completion and the actual completion time due to the lower software production rate. To solve this problem, management performs a reallocation of resources to boost software production. In this situation, they suspend the quality assurance effort for a period of time.

This effect occurs again for combined policy 9, as the sharp increase in the detected errors waiting for rework leads to a steady decline. In the same time-period the rework rate is declining to a lower level. As explained in combined policy 6, the rework team is exhausted from the previous effort. In addition, the quality assurance effort is suspended and no detected errors are sent to rework. Management have re-allocated more resources to software development, at the same time as fewer errors waiting for rework removes some of the pressure for a high rework-rate. Compared to the base-run, this phenomenon does not occur. Since there is no continuous testing, there is no sudden increase in the detected errors waiting for rework at an early stage. In addition, there is no capture re-capture effort to pick up escaped errors and feed them back to the quality assurance and rework effort.

During the period of suspended quality assurance, the increase in software production allows more tasks to be completed. The rework team is able to maintain a stable high rework rate, and the stock of detected errors decreases. Since quality assurance has been suspended, completed tasks are sent directly to testing. In addition, the increase productivity due to learning and experience enables the production team to hold a higher completion rate. The project is into the coding phase, and the simpler coding tasks are finished at a higher rate. This increase in production reduces the perceived man-days still needed, and schedule pressure is lowered. The project is also closer to completion. These factors allow management to re-initiate the quality assurance effort due to the lower schedule pressure. Due to the fact that the quality assurance effort has been suspended there is a backlog of both new generated errors and previous untagged errors that testing have re-captured and sent to the quality assurance staff. These errors are now identified and sent for rework. This is observed in the second large increase in detected errors waiting for rework that occurs in this time-period. The peak is consistent with the base-run peak, but the magnitude is higher and occurs with a delay due to the suspension of the quality assurance effort.

When we observe this effect and compare it with the graph from policy 6 it becomes apparent that the second peak is higher in this policy. The reason for this added increase is applying the client review stage. The client has now been able to test the application, and provided the testing-team with feedback on deviations that must be corrected for the software to function satisfactory for the client’s needs. These errors are now entering the stock of detected errors, and the amount of detected errors waiting for rework increases above the previously reported levels. Management is again given a signal for a re-adjustment of resources as the rework effort must obtain a higher rework-rate in order to finish the project. The result is a second increase in the rework rate to accommodate the extra errors waiting to be reworked. Slack-time is cut and schedule pressure causes a sense of urgency. There is a positive motivational factor, as the project is close to completion. When this extra rework-effort is increasing, the software development team is finished with all production tasks. Hence, the total number of remaining tasks is absolute. The rework-team can observe the actual remaining effort needed to complete the defect removal procedure, causing positive motivation for completing the job.

Since all production tasks have been completed and the staff from all efforts is slowly transferred to the final system testing, the rework effort is completed at a declining rate. There are more errors waiting to be reworked in the stock, and the remaining rework-team must finish these tasks before the final test can be initialized. The same procedure is observable in the base run, but due to the larger amount of detected errors, the effort is finished at a later stage than in the base run. The rework team perform the tasks still needed to be corrected with a steady decrease of the total manpower available for this effort. This leads to the observed behaviour in the graph at the end. Detected errors waiting for rework, and the rework rate slowly declines to zero. This concludes the analysis of policy 9.

Analysis report for completion time, detected errors reworked, man-days expended and benefit-cost ratio

The first table in this analysis is presented below and contains the completion time for policy 9 and the base-run.

Simulation run	Completion Time
Base-Run	438 Days
Policy 9	545,5 Days

Figure 152. Project completion time for base run and policy run

In the base-run there is one system-test at the end of the project after 80% of all tasks have been finished. Allocation of manpower to quality assurance is set to 15% of all manpower after training. In combined policy 9 we combined the effect of continuous testing at an earlier stage in the project when 20% of all tasks were completed, a quality assurance manpower level of 20% of all daily manpower after training, and the client review stage.

The difference in completion-time is expected, due to the factors and behaviours observed in the previous graphs above. Continuous testing and an increase in quality assurance provide a longer completion time. There are several reasons for this difference. The first reason lies in the total amount of errors detected. With a higher level of quality assurance staff, the potential to detect errors increased. The stock of detected errors waiting for rework increased in a manner that surpassed the base-run with a clear margin. The additional effect of continuous testing pushed the amount of detected errors even further. At the second half of the design phase of development after 20% of all tasks are finished, we observed a clear peak in detected errors waiting for rework. Hence, the amount of errors detected was much higher than for the base-run. The increase in detected errors yielded an increase in the rework rate. More manpower allocated for rework lead to a decrease in software development because of less manpower remain for development effort and more days were spent to complete the project.

The second factor stems from the suspension of the quality assurance for a period of time. This delayed the completion of the quality assurance effort. In addition, the development tasks completed during suspended quality assurance where directly sent for testing. This provided a back-log of untagged errors that needed to be addressed when the effort was reinitialized. This caused an additional delay, with a peak of detected errors waiting for rework after the original base-run had completed its number of tasks.

The third reason for delay stems from more allocation of manpower to quality assurance leading to less manpower remaining for production activities, and tasks development rate decreases in circumstances. The fourth reason is communication overheads and exhaustion-factors. More quality assurance effort leads to more communication. These communication procedures take time to incorporate effectively, and lead to initial delays. Also with more errors detected, exhaustion factors for both rework and quality assurance lead to a loss of productivity.

The fourth reason for a longer completion time is related to the inclusion of the client review stage. The client review stage is initiated when testing starts at 20% of all tasks completed. However, the process contains an information and process delay. When an application is ready for client-review, the testing staff prepares a presentation of it for the client staff. The client staff is then able to test the application. They review it, and report on problems or deviations that diminish the usability of the software. These deviations are then analysed by the testing-team to reveal the underlying errors causing the client dissatisfaction. Hence, the errors are fed back to the rework effort at a later stage. This causes an additional increase in the detected errors waiting for rework towards the end of the project.

The next table presented in the analysis contains the total amount of errors that were removed from the software during the software project cycle. For a policy to yield a higher level of software quality, it is important that the amount of corrected errors surpasses the base-run significantly. My aim is improved quality, so this factor is fundamental. The results are reported below for policy 9 and the base-run.

Policy simulation	Defect removal
Base run	709,40 errors
Policy 9	1356.33 errors

Figure 153. Project defect removal for base run and policy run

Comparing the results of defect-removal between base-run and policy 9 provides a significant difference. The base-run was able to detect and remove 709,40 errors, while for policy 9 the increased quality assurance staff, continuous testing and the client review increased the total defect removal to 1356.33 errors. This difference in result strengthens the argument of Jones and Bonsignour that testing and quality assurance are complimentary processes. The increased error detection potential, combined with capture re-capture and continuous testing provide a significant increase in errors removed from the end product. In addition, the client review stage increased the total defect-removal number by an additional 102.83 errors compared to policy 6 where only the increased quality assurance staff and continuous testing were in effect. This result strengthens the argument for a client-review. The client is a valuable source of information that is able to reveal additional weaknesses and errors in the software being produced. The client holds a unique position in this regard, as he alone holds expert knowledge on the work-environment that the software will be deployed under. This increase in detected errors reported above, illustrates how a client acceptance test improves software quality beyond the original base-model and combined policy 6.

The last table presented for policy 9 is the man-day expenditure. Man-days constitute the costs in a software project. Any policy that aims to improve software quality within a cost beneficial frame needs to balance the ratio between total errors removed and the total costs in man-days. For this policy we re-allocated more manpower to the defect removal effort, and the project needed more time to finish compared to the base run. This will be reflected in the man-days spent on the project as well.

Policy simulation	**Man-day expenditure**
Base run	3878,70 Man-days
Policy 9	5669.62 Man-days

Figure 154. Project man-day expenditure for base run and policy run

The results reported in the table above are consistent with my analysis up to this point. The man-days spent in policy 9 are significantly higher than for the base-run. In policy 9 a significant amount of manpower is allocated to quality assurance and testing. This factor leads to a lower potential software development rate, and the project needs more time to finish the production cycle. The second element that explains this increase in costs is the effect of detected errors on the rework-effort. When more errors are picked up at an earlier stage, the rework effort is increased to match the level of detected errors waiting for rework. Hence, management is re-adjusting more manpower and effort to the defect-removal side of the software project. This lowers the potential software productivity rate even further, and schedule pressures ensue. The rework effort peaks at a high rate, and exhaustion factors kick in. The result is a situation where management perceives the project to be severely behind schedule. The remedy for the situation is a suspension of all quality assurance procedures. Manpower is re-routed to software development. This causes a delay in production, and the total amount of manpower is adjusted upwards by management. Since the quality assurance effort has been suspended, the errors that are found in testing are automatically un-tagged errors. Hence, a back-log of untagged errors awaits identification by the quality assurance team. As a result, when the quality assurance is reinitiated the level of detected errors waiting for rework increases again and causes a second significant peak. More man-days are expended as a result.

Due to the nature or client review, the information delay causes a second peak of increase in the stock of detected errors waiting for rework. This is the third element that causes a longer rework-session. The client-review errors are fed back to rework effort in order for them to be corrected and removed. Management allocates again more manpower to the rework-effort, and as a result the man-days expended increases further. These elements influencing policy 9 and causes a significant increase in man-day expenditures compared to the base-model. In sum, the increase in software defect removal causes an increase in costs.

The next part of the analysis is the benefit, cost and benefit-cost ratio. For this policy to increase the software quality within a cost-efficient frame, the benefit-cost ratio must be significantly higher compared with the base-run. The benefit side is depicted in the cumulative detected errors reworked, and the cost side is depicted in the cumulative man-days expended on the project.

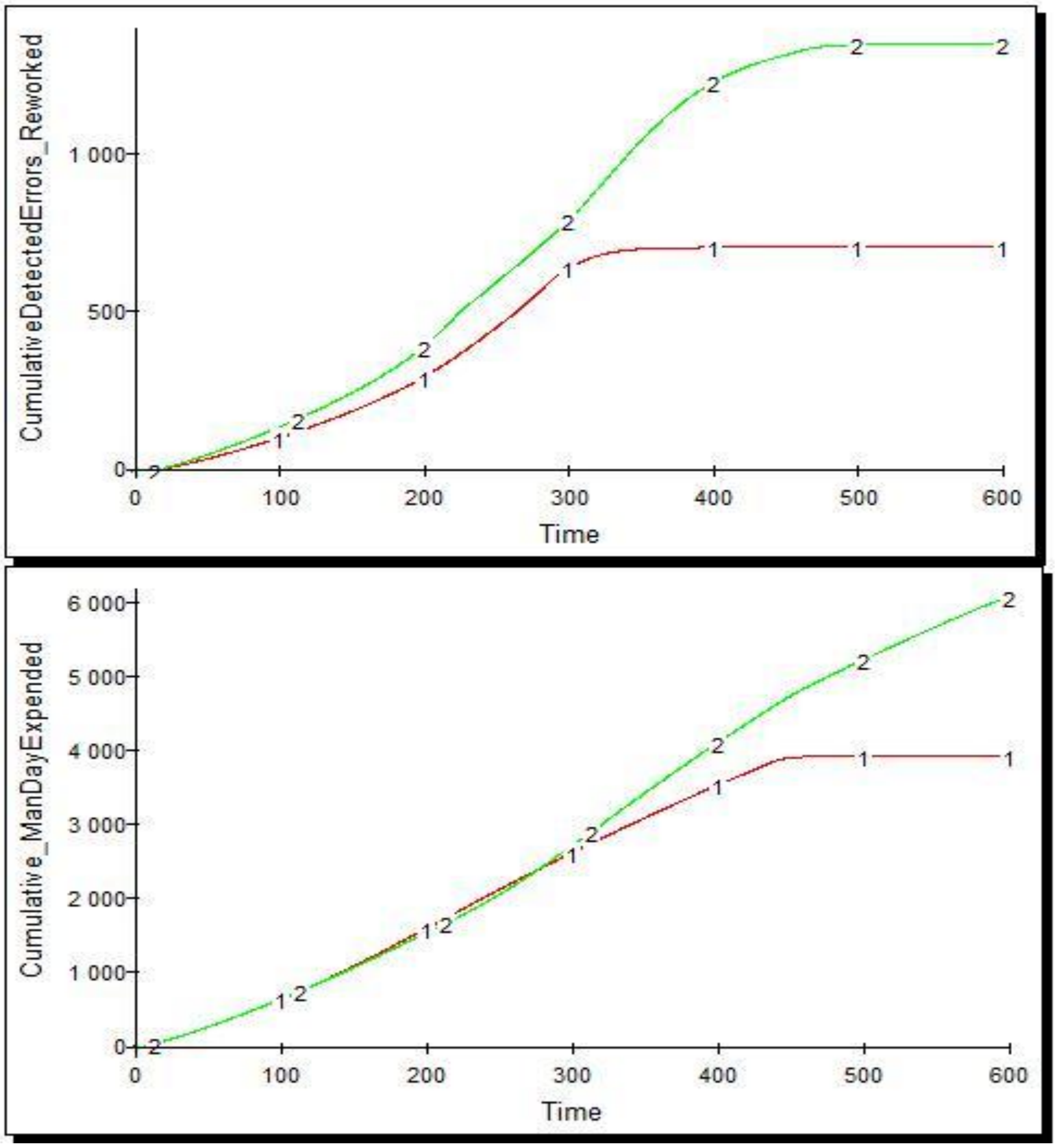


Figure 155. Cumulative detected errors reworked and cumulative man-day expended, Red line presents the base run and green line presents the policy run

From the cumulative detected errors reworked, it is clear that policy strategy 9 enabled the defect removal effort to detect and rework a significantly higher number of errors compared to the base run. The inclusion of more quality assurance at the beginning proves to be an effective policy, as the error detection potential is depending on the size of the quality assurance staff. Design errors are more fundamental, and the error density at this point is low. Therefore it requires extra attention from the quality assurance side to detect them early. The addition of a capture re-capture system allows previous escaped errors to be returned for quality assurance from the testing effort as well. The result is a clear increase in the number of detected errors that are reworked.

From the initial start the number of reworked errors is higher than for the base-run. When the testing regime is activated, there is a clear boost in the cumulative detected errors reworked as the testing effort is able to pick up both escaped errors and bad fixes. These are then fed back to the quality assurance and then rework efforts and corrected. For this policy we also introduced the client review, and this holds a significant effect as well. From the graph we observe that the total amount of cumulative detected errors reworked is substantially higher than for the base-run. Towards the end of the project the cumulative detected errors reworked is increasing to a higher level than observed in earlier policy analyses in this chapter. This is due to the client review, and it is consistent with the empirical reported earlier in this paper. Client reviews are valuable tools for detecting and removing additional deviations from the end product.

On the cost-side we observe that the expending for man-days for the project is higher than for the base run. This is consistent with the reported completion time. More effort for defect removal due to a higher rate of error detection, expended more man-days than the original base-run in order to complete the project within a reasonable time-frame. This caused additional costs for the project, and the graph depicts the significant difference in the man-days expended between policy strategy 9 and the base run. There is a significant improvement in detected and reworked errors, but the effort has increased the man-day expended as well.

In regards to the behavioural pattern of my analysis, we observe the expected S-shape. The initial behavior for cumulative detected errors reworked, and man-day expenditure is an increase as activity in the project initiates. The result is an initial behavior of exponential growth. However, as the project moves towards completion the behavior pattern changes. The goal for the project is zero remaining tasks. When the project progresses towards this goal, the level of detected errors reworked and man-day expenditure levels off towards the goal.

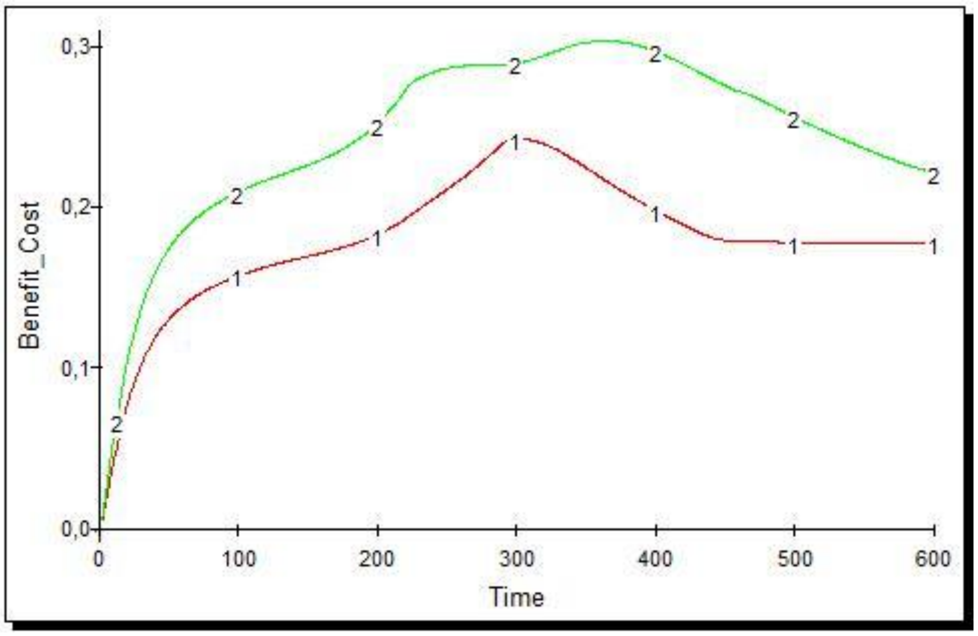


Figure 156. Benefit-cost,
Red line presents the base run and green line presents the policy run

The final graph is shown in figure 156 depicted for this analysis is the benefit-cost ratio. The graph above depicts the relationship between benefit and cost for the project, and the base-run. From the start it is apparent that the increase in defect-removal yield a beneficial gain compared to the cost in man-days. From the beginning of the project, the benefit-cost ratio is higher in policy strategy 9 than in the base model. The larger quality assurance team is able to detect more errors at an early stage, and these are reworked at a higher rate. The effect is a positive increase in the benefit-cost ratio that is at all times higher than for the base run.

The graph also depicts the significant increase in the cost-benefit ratio provided by the continuous testing, as the ratio peaks to a higher level after its initiation. It’s also interesting to note, that suspending the quality assurance effort caused the benefit-cost ratio to level out. This policy may improve software production, but could harm benefit-cost if it is suspended for a longer time-period.

The final important observation from this graph is the final peak in the benefit-cost ratio. The base-run reaches its peak at a lower level, when the software productivity rate is high and the tasks are completed faster. After all tasks are performed the benefit-cost ratio for the base-run lowers in level. At this stage in time, the client review is providing more errors to rework. The result is an additional increase in the benefit-cost ratio. The client review is not an expensive effort for our project, since the client provides the staff-level allocated from their side to the effort. Therefore these errors are found at a lower cost for the project, and the benefit-cost ratio increases.

In sum, the additional errors detected by the continuous testing, a high allocation to quality assurance and the client review process are all beneficial for the software project. The ability to remove errors provides better quality of process and end-product quality. The extra time spent is not significantly higher compared to the average record from the industry, where most projects are delayed by an average of 33.4% (Jones & Bonsignour, 2012). Hence, policy strategy 9 is able to provide better software quality within a cost-beneficial frame.

8.1.3.6 Combined policy 10: Synthesis of policy 3,1,2 and 4.

The final synthesized policy combines the four policies we have analyzed so far. This policy analyses the effects of all policies concerning the effect of manpower allocation for quality assurance, testing and rework efforts and finally applying the client review effort. My analysis has shown so far that there are two viable options for the rework desired delay that yield an improved benefit-cost ratio. By setting the desired rework delay to 10 days or 20 days rather than the original 15, leads to more or less allocation of manpower to rework effort and both strategies provide a better benefit-cost ratio due to the advanced and continuous testing effort, larger quality assurance effort and the capture re-capture process. This final analysis includes the effect of the client review stage, and the influence it holds on the ability to remove errors and provide better software quality.

The original base-run is set to the normal formula used in all my analyses so far. Through the "fraction of effort for systems testing" management allocates manpower for testing near the end of the project when 80% of all tasks have been completed. This provides an original scheme of a single system test. The normal managerial strategy for allocation of manpower to rework effort through the rework desired delay-time is 15 days, and the planned fraction of manpower for quality assurance is determined in the original model to hold a level of 15% of total daily manpower after training.

For the combined policy there are two policy strategies that will be analyzed. In policy 10 strategy 1, more manpower is allocated to rework effort through setting the desired rework delay to 10 days, while in policy 10 strategy 2 more manpower is allocated to rework effort through setting this variable to 20 days. For testing we deploy for both policy-strategies the continuous testing-regime that initiates earlier in the project, in the middle of design phase of development, after 20% of all tasks have been completed. The previous analyses of the impact on quality from the quality assurance effort leads us to incorporate the increased level of 20% of the total daily manpower after training as planned fraction of manpower for quality assurance, compared to the original 15%. The graphs below in figure 157 depict the results from both strategies 1 and 2. The red line with “1” is the base run, the green with “2” is the strategy 1, while the final blue line “3” is the strategy 2.

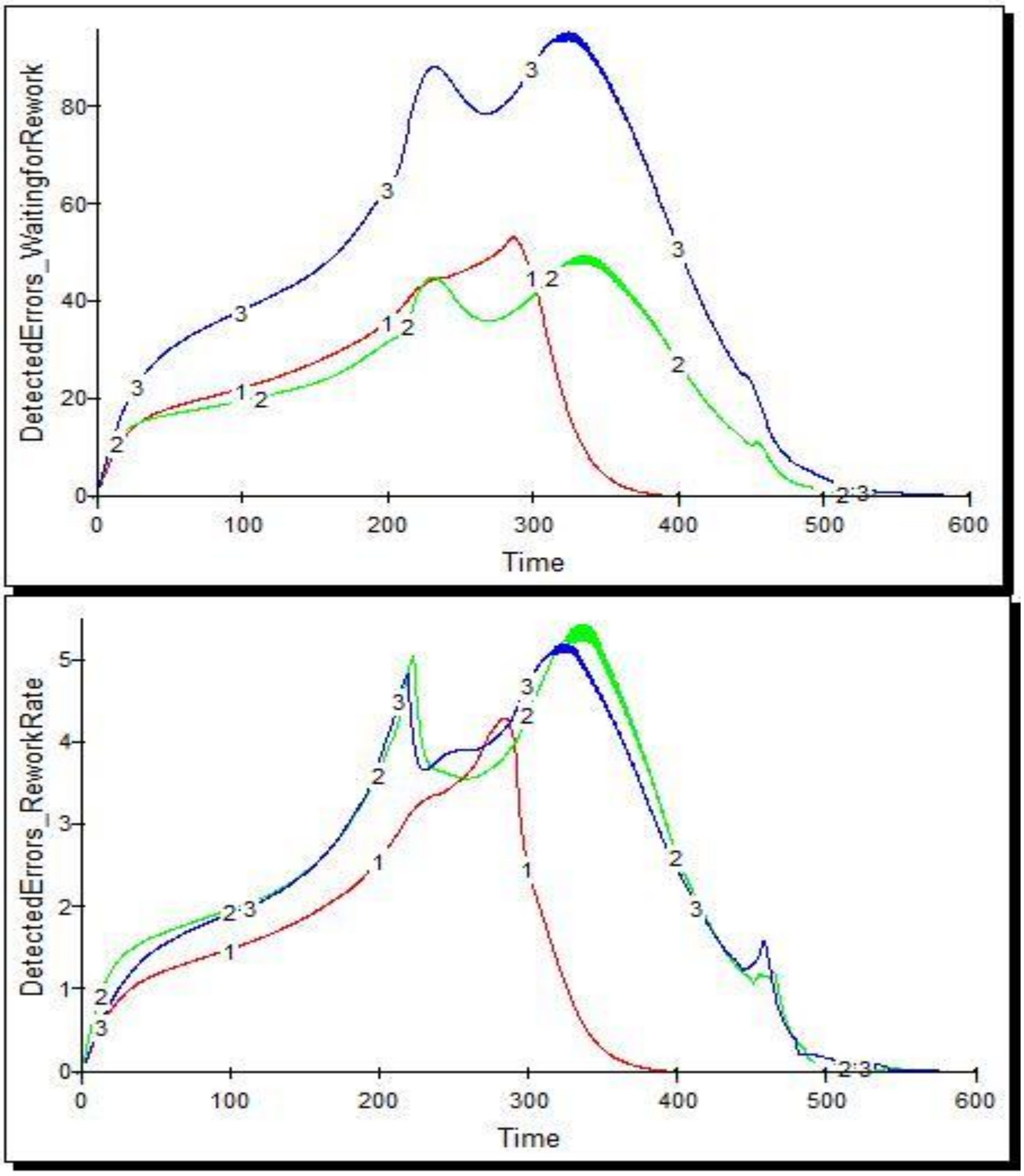


Figure 157. Detected errors waiting for rework and detected errors rework rate, Red line presents the base run, green line presents the strategy1 and blue line presents strategy2

<u>**Policy 10 strategy 1**</u>

The software project starts up, and the software production team produces tasks that are sent to quality assurance. In this strategy we have a larger quality assurance staff and a larger rework staff through a shorter desired rework delay. As in earlier analyses the increased quality assurance manpower is able to detect more errors. This is due to the higher potential error detection. This effect combined with a decrease in the desired rework delay is evident in both detected errors waiting for rework, and the rework rate.

The amount of detected errors waiting for rework increases at the same rate as the base-run. However, soon the allocation of more staff to rework leads to a lower level of detected errors waiting for rework compared to base run. This is not due to lesser errors detected, but rather an effect of the increased rework rate. In the second graph in figure 157 we observe how the rework rate holds a significant higher level than the base run. For a period of time the increase in rework rate follows a steep pattern. However, as described earlier in combined policies 5, 6, and 7, the rework effort lowers in intensity due to human factors. The level of fatigue, motivational losses and stress lowers the rework rate. The same effect influences the development and the quality assurance staff, and the level of detected errors waiting rework increases at a slower pace. The behavior of the graph of detected errors waiting rework follows that of the base run at this point in time, but at a higher level. However, as the testing regime is activated, the change in behavior becomes apparent.

After 20% of all tasks have been completed, in the middle of the design-phase, the testing effort is activated. This leads to a clear increase in detected errors waiting for rework, and the rework rate. The testing effort identifies tagged previous fixed errors and untagged previous escaped errors that are returned for either quality assurance or rework. The increased quality assurance team is able to identify and send more new generated errors and previous escaped errors to rework, as well as bad-fixes are added to the stock of detected errors through the testing effort. The effect is a sharp increase and peak in the detected errors waiting for rework. The increase in detected errors for rework signals to the management that more effort is needed for rework. In addition, the rework staff has now been able to recover from the initial exhaustion. Due to learning and experience, and the simpler nature of rudimentary design-errors the rework rate increases sharply. In addition, the software production rate is higher at this point in time due to simpler coding tasks, a gain in experience and an increase in staff productivity. So more tasks are developed and the total amounts of errors increase as the amounts of tasks developed increase. This sharp increase is not evident in the base-run, as it is fuelled by the effect of continuous testing.

As in the previous combined policies, the large amount of errors detected and reworked influences management’s perceived man-days still needed. This leads to an increase in schedule pressure, and a gap between scheduled completion time and actual completion time. In this case, the effect is even more severe. This is due to the fact that the increased allocations for rework, testing manpower have left fewer resources for software development.

The result is a suspension of the quality assurance effort as described in combined policies 6 and 9. Due to exhaustion, the rework effort is reducing its rework rate. With a suspended quality assurance effort there are no new errors detected and sent to rework. By suspending the capture-recapture process, all errors picked up by the testing effort are un-tagged and miss counting as previous escaped errors are fed back to quality assurance. This leads to a situation where a back-log of detected errors ends up in the quality assurance department rather than detected errors waiting for rework. Compared to the base-run this behaviour is not following the base run. In the base run, the testing-phase is not active until the very end of the project. The manpower allocated for quality assurance is smaller, and the desired delay for rework is longer and so rework manpower is smaller. Therefore the sharp increase in schedule pressure leading to a suspended quality assurance effort is not present for the base-run.

After a period of suspended quality assurance, the effort is reinitialized. The effect on the detected errors waiting for rework, and the rework rate is imminent. The back-log of untagged errors are now identified and sent to rework at a high rate. In addition, the project has now entered the second half of the coding phase. More tasks have been produced, and the production rate is higher. Just as in the base-run towards the end of the project more errors are both found and reworked. However, due to the delay from suspending the quality assurance effort, the peak occurs later in this policy 10 strategy 1. The suspension of quality assurance also holds a rubber-band effect, as the increase in both detected errors waiting for rework, and the rework-rate returns to the previous peak level.

In addition to this observed increase in detected errors, we see that the peak occurs at a higher point than in the combined policy 7. This extra addition of detected errors stem from the client review effort. The client has now tested the software, and provided the report on deviations. These deviations are significantly diminishing the software’s usability for the client’s needs. These errors are identified and analyzed by the testing-teams and sent back for rework. The result is an additional increase in the detected errors waiting for rework, observed in the graph in figure 157. Suspending the quality assurance effort and the additional entry of errors from the client review causes a higher second peak than in combined policy 7. Compared to the base-run it occurs later due to the delay from suspending quality assurance and the higher error detection level.

Since all production tasks have been completed and the staff from all efforts is slowly transferred to the final system testing, the rework effort is completed at a declining rate. There are more errors waiting to be reworked in the stock, and the remaining rework-team must finish these tasks before the final test can be initialized. The same procedure is observable in the base run, but due to the larger amount of detected errors in policy 10 strategy 1, the effort is finished at a later stage.

Policy 10 Strategy .2

The software project initiates, and tasks are being controlled by the quality assurance staff. In policy 10 strategy 2 the quality assurance staff holds a higher manpower-level than for the base-run. However, the desired rework-delay is set for a longer time-period of 20 days rather than the original 15, which leads to less manpower allocated for rework effort. The effect of this difference is immediately apparent as compared to the detected errors waiting for rework in both the base-run and for strategy 1.

The increased quality assurance staff is able to detect more errors, as their error detection potential is high. The rework-effort is slower to initiate their effort as the rework-delay is longer. Therefore we observe that the number of detected errors waiting for rework increases drastically. When we observe the rework-rate, the increase in detected errors initiates a strong rework-effort. The initial response from the rework-team is high due to positive motivation and a rested work-force. However, due to the size of the rework-team and the longer delay it is not able to maintain as high a rework-level as for strategy 1 that held a shorter desired rework delay and thus more manpower were allocated for rework. Compared with the base-run, the rework effort holds a higher level. This is due to the sense of urgency and schedule pressure derived from a sharp increase in detected errors. This initial boost in the rework rate is not maintained for a long period of time, and we observe that the rework rate decreases to a lower level. This is due to exhaustion from the demanding design errors, motivational losses and fatigue.

The level of detected errors waiting for rework also shows a lower level of increase. This is due to the same factors as explained above for the rework-team. Loss of motivation, fatigue, and communication overheads influences the development and the quality assurance staff.

Design errors hold low error density, so error detection requires more effort. In addition, communication losses reduce the productivity of development and quality assurance. As explained in combined policy 7, the reduction in the detected errors waiting for rework is also linked to manpower allocation. With more manpower allocated for quality assurance, less manpower is available for software production. Hence, fewer tasks are completed and fewer errors are detected by quality assurance.

After 20% of all tasks are completed, the testing effort is activated. This leads to an increase in the detected errors waiting for rework as well. In addition, the effect of learning and relatively simpler design-tasks for production to accomplish provides a higher software production rate. As a result more tasks are accomplished, and more mistakes committed. The combined effect of testing and high error detection potential from quality assurance, increases the stock of detected errors waiting for rework to a high level. This signals again a need for an increase in the rework-effort, and this is observable in the rework rate. Compared with the base-run, this sudden increase in both detection of errors and rework rate break with the original behavioural pattern. The reason for this is explained through the testing effort.

As observed in strategy 1 the increase in detected errors waiting for rework through allocation of more manpower to quality assurance that leads to less manpower available for software production, causes high schedule pressure. The response to this development, as explained in strategy 1, is to suspend the quality assurance effort for a period of time. This is observable in the detected error stock, as the number of detected errors waiting for rework drop significantly. This reduction occurs at the same time as the rework effort decreases significantly. The exhaustion from the effort combined with motivational losses and turnovers, leaves the rework staff unable to maintain the high level of rework. In addition less detected errors waiting for rework leads to downward managerial adjustment for rework manpower allocation.

The behaviour displayed by strategy 2 is different in this situation than strategy 1. Quality assurance is suspended, and more manpower is allocated to the software production effort. However, the total amount of detected errors still waiting for rework signals for management that the effort must be maintained at a high level. Hence, the rework rate that dropped to a level close to the observed level for strategy 1, increases again more rapidly. This increase is also higher than compared to combined policy 7. The client review stage is providing more errors as feedback for rework, and this holds the same effect as for strategy 1. The result is a second peak that is delayed as compared to the base run, due to the suspension of the quality assurance effort, while the additional errors from the client review push the level to a higher peak. Compared to the base-run, the amount of detected errors for the final peak is significantly higher. The software production effort has finished all tasks at this point in time, and as observed in strategy 2, the manpower is slowly moved to the final testing effort. Due to the higher amount of errors detected, the time for the rework effort to correct all detected errors is longer than compared to the base run. This concludes the analysis of policy 10 strategy 2.

Analysis report for completion time, detected errors reworked, man-days expended and benefit-cost ratio

The first table presented is the completion time of the project.

Simulation run	Completion Time
Base-Run	438 days
Policy 10 strategy 1	549.5 days
policy 10 strategy 2	541 days

Figure 158. Project completion time for base run and policy strategies runs

The results from the table are in line with the observed values from the two graphs presented above in figure 158. Due to a higher level of error detection, more manpower allocated to the defect-removal effort, and the additional client review stage, the project ran for a longer time-period. In the base run the project is completed in 438 days. For strategy 1 the project was completed in 549.5 days. For strategy 2 the project was finished after 541 days. The difference here can be explained by the higher error-detection level for strategy 1 compared to 2. Both completion times are within tolerable levels compared to the data from the software crisis. The completion-time is within the average reported delay of 33.4%. The elements that cause these increases have been thoroughly covered through my analysis. The increase manpower allocation to rework, quality assurance, testing, and client review all have impact on the completion time. More manpower to this effort leads to a reduction of the potential software development rate. In addition the increase in detected errors spark an increase in the rework rate, and more resources are allocated to the defect removal effort. These factors have been explained in detail through combined policy 5,6,7,8, and 9. The 10th policy is the combination of all the combined policies and the client review effort.

From this analysis it is also possible to reveal the effect of the included client review stage. In my previous analysis of the client review stage as an isolated effort, the completion time for a project incorporating the client review was shorter than a similar policy-combination without the client review. This was observed for combined policy 8, and it is observable for combined policy 9 and 10 as well. The client review stage is a continuous effort in the same way as the enhanced testing effort. This provides management with better information to the perceived time still needed to complete the project. The available information on errors provided by the enhanced testing effort and the client review allows management's perception of the rework-rate needed, to be more accurate. Compared with combined policy 7, the completion time for policy 10 is earlier. The only difference between policy 7 and 10 is the client review stage. The same element is true for combined policies 9 and 6. With a client review there are more errors detected, but less days needed to complete the project. This strengthens the explanation provided in policy 4 regarding this phenomenon.

The second table that is provided in this analysis is the total amount of errors that have been detected and reworked through the software production cycle. It is important that any new policy-strategy can help management to obtain increased software quality. Part of this effort is accomplished by the total amount of errors that are removed from the software before it's published to the client. The next table reports the total amount of errors removed.

Policy simulation	**Defect removal**
Base run	709,40 errors
policy 10 strategy 1	1370.97 errors
policy 10 strategy 2	1344.79 errors

Figure 159. Project defect removal for base run and policy strategies runs

From the table above we observe that for the final policy simulation there is an expected increase in the total amount of corrected errors. In policy strategy 1 and 2 we are able to detect and rework a significantly higher number of defects compared to the base model. From my analysis of the combined policies so far, this result is in line with the previous analyses. The positive effects of enlarged quality assurance, continuous testing effort and the client review are providing a level of defect-removal that is unsurpassable by any other policy-combination in my model. The desired rework delay does not hold a direct impact on the total level as described in detail under policy 1,5 and 7. Therefore, the difference in error detection between strategy 1 and 2 in combined policy 10 is not huge. However, by cutting the desired rework delay to 10 days rather than 20 days as in strategy 2, 26,18 additional errors were detected. Hence there is an improvement in defect-removal favoring strategy 1.

The final table consists of the man-day expenditures for both strategies 1 and 2. The man-day expenditure reveals the differences in cost-level between strategy 1 and 2, and is comparable to the base-model. The base model expended 3878,70 Man-days during the software project. For policy strategy 1 and 2, it is important that the level of man-days expended is not outweighing the increase in defect removal.

Policy simulation	**Man-day expenditure**
Base run	3878,70 Man-days
Policy 10 strategy 1	5693.31 Man-days
Policy 10 strategy 2	5638.26 Man-days

Figure 160. Project man-day expenditure for base run and policy strategies runs

In the final combined policy we deploy all the defect-removal enhancements that I have incorporated simultaneously. The results in man-day expenditures above are consistent with the effect we have observed from each isolated policy and combined policy. When manpower has been moved from software development to defect removal, the completion-time for the project as a whole is longer. There are several elements that influence this development. One element is the increase in quality assurance staff. The increase in QA staff leads to more detected errors early in the project. This leads to an increase in the rework-effort that holds a high level. This increases the man-day expenditures from the beginning of the project. The continuous testing process initiates after 20% of all tasks are completed. The additional manpower allocated to this effort reduces the potential software development productivity, and the project needs more time to finish. The testing effort detects and capture errors that are sent back to quality assurance and rework through the capture re-capture system. This increases the level of the rework-effort to a significantly higher rate. Schedule pressure and elevated rate of detected errors waiting for rework leads to a managerial re-adjustment that increases the total manpower for the project. The elevation also causes the quality assurance effort to be suspended for a period of time. The errors detected in the client-review stage are now being sent back to rework as well. This leads to more adjustments in both rework-effort and manpower-allocation. All of these factors lead to an increase in man-day expenditure, and not surprisingly this is the combined policy that reports the highest expenditure of man-days compared to the others. The remaining question is the benefit-cost ratio.

The benefit and cost development is depicted on the next two graphs in figure 161. The benefit for the policies is measured by the cumulative detected errors reworked. The more errors that have been detected and reworked translate to a higher degree of software quality and process quality. On the cost side I report the cumulative man-days expended on the project. In order to calculate the benefit-cost ratio these two elements must be taken into consideration.

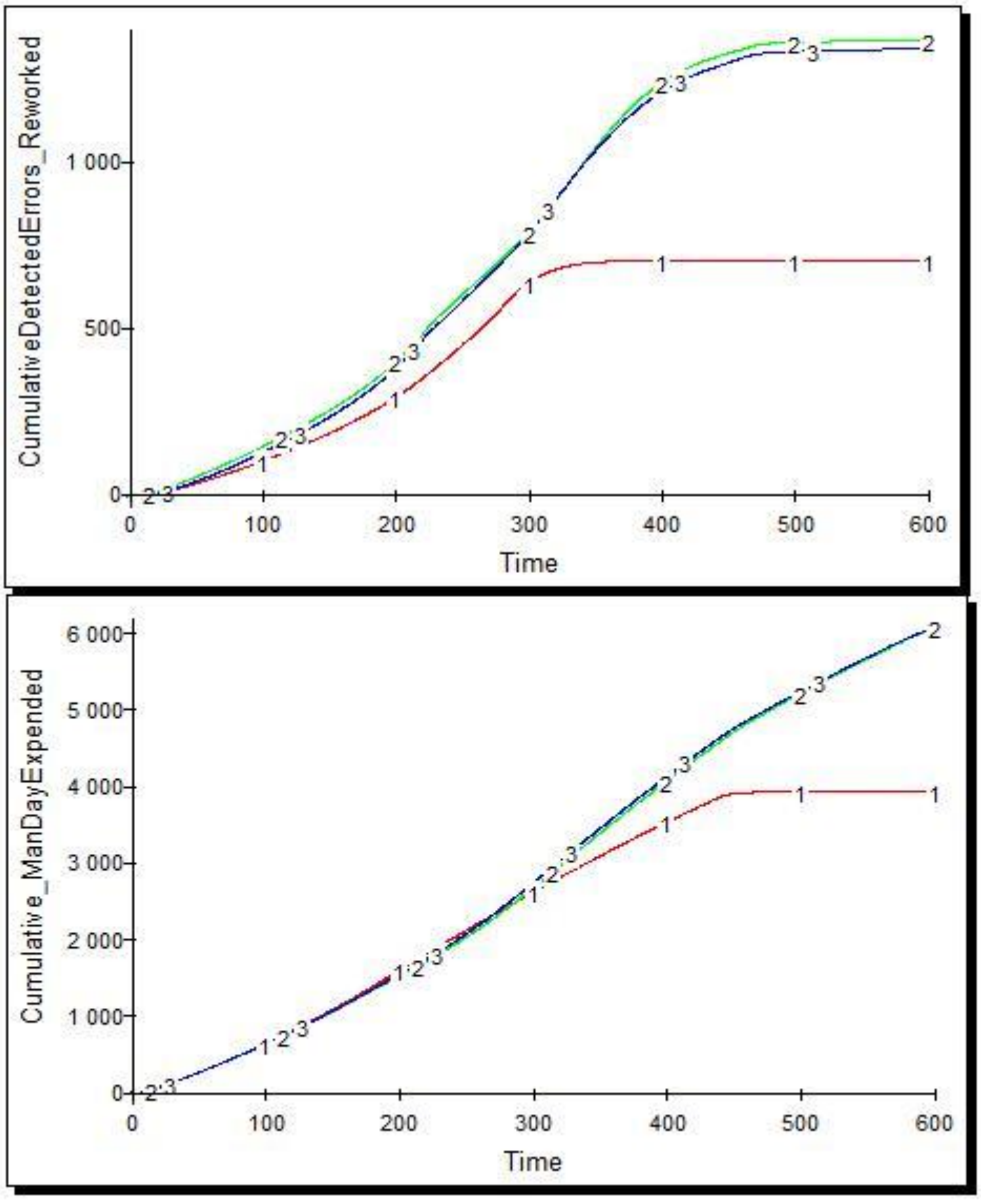


Figure 161. Cumulative detected errors reworked and cumulative man-day expended, Red line presents the base run, green line presents the strategy1 and blue line presents strategy2

The first graph in figure 161 represents the cumulative detected errors reworked by both strategy 1 and strategy 2 as the benefit of the project. The graph of cumulative detected errors reworked shows the significant impact of increased quality assurance staff, continuous testing, and the client review stage. For both strategies the number of detected errors reworked are higher than the base-run by a clear margin. The base-run is able to remove nearly 709 errors, while both strategy 1 and 2 removes close to 1370 and 1344 errors. This is a significant difference that will boost software quality. In addition, the client review has enabled the end-user to provide additional insight to the software product's quality. By including the client I have been able to remove more errors, and ensure that the software is functioning satisfactorily to the client's needs. This is a valuable addition to this effort that will provide that important end-product quality. The question that remains is the benefit of additional defect removal compared to the increase in costs.

The second graph in figure 161 represents the cumulative man-days expended in the project by both strategy 1 and strategy as the cost of the project. We observe that the total man-days expended on this effort are significantly higher than for the base model. Due to more manpower allocated for defect removal, software production held a slower rate than the base-model. In addition, the pressures from increased quality assurance and errors waiting for rework caused a situation where quality assurance had to be suspended. This delayed the project further, and caused additional manpower-changes. Management had to hire more personnel in order to accommodate the new level of rework needed, at the same time as software production had to be maintained. For these to policy strategies to be valuable, the benefit to cost ratio must be higher for both policies.

The final two graphs for the development of cumulative detected errors waiting for rework, and the cumulative man-days expended displays the characteristic S-shaped behavior pattern. The initial development follows that of exponential growth for both base-run and policy strategy. However, the difference in allocation changes the level for the policy strategy compared to the base-run. As the project progresses the end, it reaches its goal of zero tasks remaining. At this point the increase in man-days expended and detected errors waiting for rework levels off towards the goal. This provides the S-shaped pattern described in detail in the beginning of this chapter.

The final analysis is the benefit-cost ratio. For the policies to improve the quality of the end-product within a cost-beneficial frame, the increase in errors detected and removed from the software must outweigh the increase in expended man-days. This is analysed in the graph in figure 162.

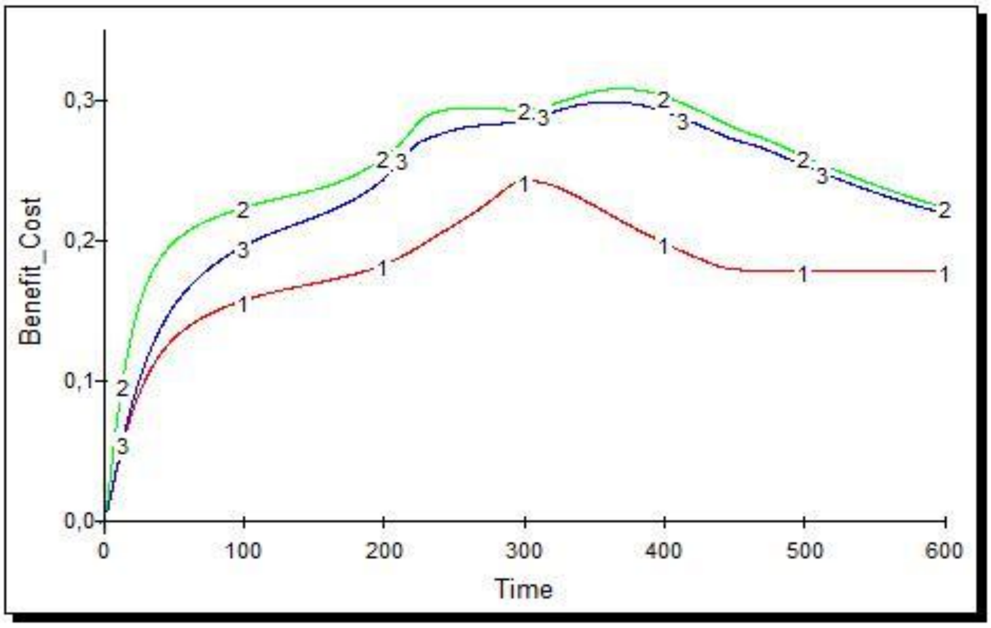


Figure 162. Benefit-cost ratio,
Red line presents the base run, green line presents the strategy1 and blue line presents strategy2

The graph above provides the benefit to cost ratio, and it is apparent that for both strategies 1 and 2 the yield is high. Compared to the base run, the benefit-cost ratio is significantly higher for the combined policy. More quality assurance provides a high benefit to the detected errors in the crucial design phase, and enables them to be corrected at an early point in time. This is important for the project, and we observe the effect in the benefit-cost graph.

The behaviour of the graph is similar to the one observed in combined policy 7. However, here we can measure the impact of the client review. Since the cost of the client review, that is the amount of man-days spent by the client, is taken by the client, the additional errors found are provided without an increase in project costs. The result is a benefit-cost ratio that is higher than for combined policy 7. The overall improvement of benefit-cost ratio higher as a direct result of the client review stage. This fact supports the argument of Hjertø and Deming, stating that the client is a vital source for quality. Any software program that is not working according to the wishes of the client is a failed piece of software. In this combined policy the client is able to aid the defect removal effort, and to provide a better base for end-product quality. This is achieved with a better benefit-cost rate than in Abdel-Hamid's original mode (Hjertø, 2003) (Deming, 1981) (Abdel-Hamid, 1991).

8.2 Concluding remarks drawn from the policy analysis

Through this chapter I have conducted an analysis of ten different managerial strategies in order to investigate the effect of the defect-removal efforts on the software development project. These analyses have revealed valuable information and strengthened the literary arguments in favor of focus defect-removal to ensure software quality. In this final part of the chapter I will highlight some of the important patterns discovered, and some of the managerial challenges to software project management.

8.2.1 Patterns from the isolated analysis

The first part of my analysis contains the isolated analysis of the manpower allocation strategies proposed for the defect removal effort. The difference in allocation of manpower to quality assurance, rework, testing and the client review all revealed how they all affected the ability to detect and remove errors. For quality assurance, the increase in manpower allowed the potential error detection rate to increase above the level of the base model. Reducing the same manpower to a lower level yielded a bad result which lead to fewer errors detected and more man-days spent as undetected errors slipped by to the testing stage.

In the case of the rework-manpower, the desired rework-delay did not yield a large difference in the ability to detect and remove errors. The rework-effort concentrates on the correction of detected errors. The difference in allocation did have an impact on the completion time. In the isolated analysis, a longer delay yielded a negative result compared to the base-run.

The isolated analysis revealed that the testing effort is the paramount effort for defect removal. The inclusion of a continuous testing system yielded significant results that surpassed the base-model in terms of detected errors reworked. From the graphs depicting the detected errors waiting for rework, and the rework rate, the impact policy 3 strategy 4 yielded the clearest change in level and behavior compared with the base model. This policy contained the managerial decision to start the testing effort when 20% of all tasks were completed.

The client review was then added to policy 3 strategy 4 in the final isolated analysis. The client review stage can only be analyzed in a project with a continuous testing regime. In my research, the aim is to improve software quality before it is published to the client. In the original base-model this would have been impossible as the project contains only one final systems test. In that model a client review would have been conducted after the final product had been produced and tested. Any changes provided by a client review would lead to post-production changes that are outside the model boundary. I therefore chose to use the policy strategy 4 from the testing effort since it yielded the highest improvement of the analyzed strategies. The result by adding the client review was an additional increase in error-detection and more errors were removed from the final software product as a result.

The isolated tests alone yield interesting results. However, as described by Abdel-Hamid in his original thesis, the challenge for software management is the complexity in a software project. The managerial decision in one department will lead to changes in another department. The important variables are interlinked, and to fully manage a system one need to understand the effects of a decision on the system as a whole. For example in the isolated analysis of the desired rework delay, strategy 2 with a longer delay yielded a negative result compared to the base-model. This would lead us to conclude that policy 1 strategy 2 was not able to improve the overall behavior of the system and lead to a higher software quality rate. When we include this strategy for our combined analyses we observe that policy strategy 2 in combination with increased quality assurance, testing and client review yield a significant positive result compared to the base-model. This is one of the important strengths of a system dynamics model. Unlike a static model, the dynamic model is able to simulate the combined influence of all variables and decisions simultaneously on one-another. The complexity of variable-relationships may prove that counter-intuitive solutions do yield a positive result due to the relationships and feedbacks of a system. The example above is one illustration of the importance for analyzing the policy strategies combined.

8.2.2 The important patterns from combined policies on defect removal and benefit-cost

The combined policies revealed a clear pattern in terms of defect-removal and the benefit-cost ratio. The most influential policy-decision is the continuous testing effort. As in the isolated analysis, the decision to incorporate a testing effort that starts when 20% of all tasks are complete propels the error detection rate upwards. In combination with other policies such as increased quality assurance, the error detection rate was significantly higher than the base-run.

For the combined analysis, both the rework strategies with a shorter and longer desired delay yielded an increase in the defect removal effort. More errors were found and removed regardless of desired delay. The difference in strategies did provide some valuable insights. For example, the schedule pressures and exhaustion of the rework-staff was a clearer problem for rework strategy 2. Longer delay and a smaller staff led to an increase in the rework-rate that matched the rework rate of policy 1. This lead to an exhaustion of the effort, and the rework rate fell drastically afterwards. Such policies could lead to an increase in turn-overs as a result of an increased work-pressure over time. The software industry holds one of the highest turn-over rates, and understanding how to avoid such situations is important for software project managers. Hence, my analyses here are an important aiding-tool for management in order to foresee such problems ahead of time.

The importance of my analyses as a tool for sounder management decisions is captured by the client-review effort. From the isolated analysis of policy 4, and the reoccurring pattern in policy 8,9 and 10, we observe that the client review provides management with additional information that allows the project to finish earlier than in similar policy-constellations void of the client review. This is an example of a counter-intuitive effect. The inclusion of a client-review increases the total amount of errors detected. Due to the information and process delay these errors are fed back to the rework effort as the project is finishing the final production tasks. The client-review continues for a while after all production tasks have been completed. In light of the other policies we would expect the completion time for the policy strategy to be longer due to these factors. However, the opposite is occurring. Due to the fact that the client review effort is able to provide management with a better perception of the rework effort still needed, the allocation is more accurate. This is described in detail under policy 4 and later in policy 8 and 10. This example illustrates the strength of dynamic models, revealing effects that would not have been discovered by a static analysis alone.

When it comes to defect removal and costs, the analysis provides a clear pattern. When defect removal is improved, and more manpower is allocated to this effort the completion time is longer. Since less manpower is available for software development, and the increased level of detected errors increase the rework-effort, the software project needs more time to finish. For all my policies the total amount of errors removed are significantly higher when compared to the base model. However, the costs in man-days are also significantly higher. To detect more errors leads to more costs in man-days this is a clear pattern in my analyses. The benefit-cost ratio supports an increased focus on defect-removal. For all my combined policies, the benefit-cost ratio surpasses that of the base-model. The positive effect of finding and removing errors outweigh the additional costs. This is supported by Jones and Bonsignour's analysis. They find that software quality leads to better financial gains in terms of costs versus benefits. However, the effort to improve software quality requires increases in overall costs and time.

There is another important factor to remember in the question of costs versus man-days and completion time. These factors are merely the short-term question of costs that management must consider. This is one of the challenges for software project management. The short-term gains by lowering standards and finishing on time can cause significant problems post-production. One side of these problems is the financial side. If management only focuses on completion time and short schedules, the overall quality of the software will suffer. This will have consequences after production has been completed. Charette's paper describes an industry where delays, cost-overruns and poor software quality leads to scrapping of finished software products (Charette, 2003). The ESSU report shows how recently developed software is terminated a few months after production due to inadequate usability (ESSU, 2010). The aftermath from failed software includes law-suits, loss of consumer confidence and losses of market shares. These are direct financial damages caused by insufficient management of software development projects.

In addition to the financial problems, scrapped or low-quality software bring other unfortunate consequences beyond the monetary realm. The modern society depends on software to run essential services. When a hospital makes a medical mistake, or a nation-wide black-out occurs it is more often a result of poor software quality (Jones & Bonsignour, 2012). The Altinn-case from Norway showed how improvements in a governmental portal for public documentation lead to a situation where sensitive personal data was available for the general public. In England several economically challenged families where put in an additional hard situation, when a software error over-transferred money through a government program for economic aid. The government sought to reclaim all the ill-transferred money, which had been transferred over a long time-period (ESSU: 2010). These are the long-term issues that management must take into consideration. Short-term gains might invoke long-term losses. These losses are crucial for management to understand and calculate during the initial planning phase. It is also important that these factors are discussed openly with the client in order to provide the best possible outcome.

My analyses here are not an attempt to determine the best policy. These analyses are meant to be an aid for managerial decision-makers and clients to understand the importance of defect removal and short term costs. To enable both managers and clients to understand the relationship between scheduled completion, defect removal and short term/long term costs is an important contribution that I wish to make. I would like for my contribution to solve some of the difficulties linked to the software crisis of quality, and this model and analyses is a modest start for that improvement.

Chapter 9: Conclusion

9.0 Introduction

The final chapter of my research contains my concluding observations concerning the question of software quality, the enhanced model and challenges for future research. I set out on this effort, in order to make a contribution towards better software quality. The damages from poor software in terms of financial losses and damages due to the software crisis have been well established over the last four decades. I argue that the financial damages due to cost-overruns and delays are symptoms of the crisis and not the cause of the disease. Lack of software quality and lack of quality in software development and software project management are the chief areas causing the software crisis of quality.

In this paper I used Abdel-Hamid's integrated system dynamics model for software project management. This model is comprehensive and robust, as it has been empirically tested and validated. From the literature in the system dynamics community, the model is part of the literary tradition in this field of research. Hence, this model was a sound platform for my effort to improve software quality. Abdel-Hamid's own concluding remarks presented at the end of the original thesis encouraged future research to look into the concept of software quality, as his model had not covered this aspect of software development as thoroughly as the managerial aspects of software development and costs.

I formulated two main research questions in chapter 5 that I wished to investigate and answer in order to contribute to increased software quality and how to obtain it through improved managerial decisions. These questions will now be presented, and I provide my answers obtained from this comprehensive analysis.

9.1 Research Question 1

The first research question concerned the major assumptions inside the original model itself. Since the model was constructed and published in 1991, the nature of software and the software industry have changed significantly. Through the mid 1990s and early 2000'nds the entire IT industry has gone through a period dubbed "the computer revolution". Hence, the changes in software mechanics, complexity and execution during the last 20 years are substantial. This development was an important back-drop for my first research question:

Q1: *Are there any unrealistic assumptions in Tarek Abdel-Hamid's "Software Development Project management" model? If so, how we can modify those assumptions to make the model more realistic?*

In order to answer this question, I had to address an important omission in the original thesis. The integrated software model was well developed in the terms of stock and flow diagrams, connecting all the subsectors together for analysis. However, in system dynamics methodology the importance of causal loop diagrams have been established. In order to capture the relationship between variables and to reveal the important feed-back loops, causal loop diagrams are deployed as an essential tool. The original model did not contain such causal loop diagrams. This reduced the transparency of the model, as the stock and flow diagrams could not portray the cross-subsystem relationship between variables for each effort. This factor makes it impossible for any researcher, reader or manager to understand the feed-back processes inside the model. Therefore, my first major contribution in this paper was the construction of complete causal loop diagrams for each sub-sector, and for the entire model as a whole. This was an important effort as it revealed assumptions that where unrealistic and needed to be modified. Without the causal loop diagrams, some of these assumptions would have been undetected. From the CLDs I revealed three major areas that needed attention in regards to unrealistic assumptions.

The first major observation from the CLD concerned the concept of testing. In the original model the testing effort was confined to a single final systems test. In the CLD this assumption created an information-gap between the testing effort and rework. Errors detected in testing would never return to the rework effort. The model assumed that all errors would be detected in testing and corrected in this stage. For a modern medium ranged software project this scheme is not applicable. Due to progress in techniques over the last twenty years, and more available information regarding testing, this assumption had to be modified. Testing is one of the most important defect removal efforts. Over 90% of all software projects in the US contain a testing effort. From the available empirical data, we learned that testing in a modern medium ranged project contains a continuous testing system with multiple testing stages. The average number of testing-stages was six for medium ranged software projects. This lack of feedback from testing to the software development effort therefore constituted an information-gap that needed to be bridged. The testing effort is not isolated to the final end-process of software development. This assumption had to be modified, and it gave birth to my enhanced testing regime. This testing regime is continuous, and contains multiple testing stages.

The second assumption that my enhanced model addressed was born from the same information-gap. In the original model the concept of quality assurance and rework were regarded as the only defect-removal effort during the production stage. When a piece of software had been developed and tasks had been completed, it was sent to quality assurance. In quality assurance the task would be investigated, and errors detected. These errors would then be sent to rework for correction. Rework corrected the errors and sent the task for the final systems test. This system is not applicable for a modern software project. Ghezzi et.al (2003) identify this process as a "code-and-fix" approach, that is only applied to very simple software programs that only cover a specific function such as a calculator program for a lap-top computer. In modern medium ranged projects, the quality assurance and testing effort are deployed in tandem with one-another. The quality assurance effort provides the micro-level analysis of code, and detects errors in the coding fabric. Testing provides the meso-level, where applications are tested in order to reveal integration-errors. These two efforts are complimentary efforts and lead to better defect removal. This leads to better perception of the rework effort needed as well. To harness this effect, and improve the realism of the original model, I constructed the capture re-capture effort. This is a modern approach to defect removal, and a valuable managerial tool for measuring the total amount of errors. In addition, the capture re-capture effort not only finds errors but determines the cause. Either they are caused by errors escaping detection during the quality assurance effort, or they are bad fixes from rework. This innovative approach combines the strengths of quality assurance and testing.

The next large effort to improve realism regarded the position of the client during software production. In the original model, the client was excluded from the model. The client was only present during the pre-production planning stage, and not included for the software development cycle. From the empirical data on modern software projects it is evident that the client is incorporated through the testing-effort. There are several techniques and processes including the client in this effort. For my enhanced model I chose to construct a client-review system. This system builds on the practice described by Jones and Bonsignour (2012) as a client acceptance test. For any software to be a success, the needs of the client must be covered. In order to ensure this element, the client is allowed to test applications and review them. Any deviations or problems causing the client to experience diminished usability of the software are revealed during the client review stage. It is vital that the end product performs the tasks it will encounter in a satisfactory manner to the client's needs. This aspect was not included in Abdel-Hamid's original model. Therefore I enhanced the model's realism by adding the client-review structure to include these real-world relationships that were previously omitted.

In addition to these three major enhancements, the CLDs dealt with of a further lack of realism concerning the rework-effort and the generation of bad fixes. The rework effort was enhanced so that the effects of software productivity, daily manpower allocated to rework, potential rework rate and error density directly influenced the rework rate. The second element is the bad-fix generation rate. From the empirical data we learn that the bad fix generation rate holds an average of 7% for all software projects. In Abdel-Hamid's original model this element is governed by a constant set to 7.5%. However, in real-world projects the reported bad fix generation can be as low as 2% for successful projects and as high as 20% for failed projects. I enhanced my model so that the generation of bad fixes is governed by known influential factors such as schedule pressure, exhaustion, stress, turnovers and loss motivation. These enhancements were all important to address and modify unrealistic assumptions in the original model. The effort provides a model that is able to reflect real-world modern projects and increases the validity of my analysis. I would like to point out that none of these enhancements would have been possible without the construction of causal loop diagrams for each subsector.

9.2. Research question 2

The second research question was formulated in the following manner:

Q2: *In terms of policy conclusion (for example: distribution used to resources), are there additional policies for instance, supplementary allocation of resources to quality assurance, rework and testing efforts that would improve the model behaviour?*

From the previous chapter I have been able to reveal some important patterns in regards to this research question. With the enhancements of the model, the policy analyses revealed the importance of allocation to obtain increased software quality and quality of process.

The most important policy is the allocation of resources to a continuous testing effort. The ability to test applications at an early stage is a prime engine for error detection. The policy analyses in chapter 8 reveal that no other defect-removal effort is able to increase the level of software quality as the continuous testing effort.

The second important discovery in my analyses concerns the quality assurance effort. Allocating more manpower to the quality assurance effort provides a higher rate of error detection. The increased allocation to the quality assurance effort allows more errors to be detected at an earlier stage. This is important for the managerial decisions towards rework and the rework effort. The combined analysis of both testing and quality assurance reveals the strong potential for increased software quality when the efforts are run in tandem with each other. The effect of the capture re-capture system allows more errors to be detected, correctly identified and reworked in a proper manner.

The client review stage holds two important features. Firstly, the increased effort towards removing errors leads to a higher error detection yield than any combined policy without the client review. In addition, the client review provides additional information to software management concerning the true rework-effort needed. This information allows management to allocate manpower to the rework-effort more accurately. The result is a reduction in completion-time for policy strategies that deploys the client review, compared to strategies without this stage.

The analyses presented in chapter 8 all provide a common pattern for defect removal and software quality. By including an enhanced testing system, the capture re-capture procedure and the client review, I am able to achieve a higher defect-removal rate than the base-model of Abdel-Hamid. However, the increase in defect removal holds additional costs as well. When manpower is allocated towards defect-removal efforts, the resources available for software development are diminished. This causes longer production time, more man-day expenditure to accommodate the increase in detected errors and the hiring of additional staff. The benefit-cost ratio is higher for my policy strategies compared to the base model. In addition, the empirical data on the software crisis describes a situation where long term costs are a continuous problem. Post-production maintenance, faulty software and litigations are all hallmarks of the long-term cost for software projects. My analyses are not meant to be a blueprint for successful software projects. Every software project has its own limits, scope and budget. My contribution is to provide a model that can work as a helpful tool for both managers and clients to understand the options for the software development project. The question of resource allocation is split between the demand of production for a deadline, and the overall quality of the product delivered. It is a balancing-act that requires both management and clients to fully appreciate and understand the relationship between both the defect-removal effort and the software development effort. All software projects are inherently complex, and the biggest managerial challenge is to manage and maintain all efforts simultaneously. In addition many software project problems stem from miscommunication between project management and clients. My analyses show that an increased focus on software quality requires better quality of process. If management and client agree on these fundamental principles before the software production stage, the process will be transparent and predictable for both parties involved.

My combined policies also reveal some dangers that management can avoid ahead of time. For example, the schedule pressures under an increased quality assurance effort leads to a suspension of this effort. This is a problem that can be avoided if management understand the underlying reasons for this phenomenon to occur. My model is also able to explain how bad-fixes are caused by schedule pressures, stress and loss of motivation. In addition, the allocation of manpower to rework and the desired rework delay hold a clear impact on the total amount of schedule pressure and work-exhaustion due to staff size. The software industry holds one of the highest turnover rates compared with other industries. The policies I have proposed here may also help management to understand how human factors such as motivation, schedule pressure and stress fuel this trend. In my mind the best remedy for the software crisis are realistic and foreseeable plans. It is impossible to plan for every contingency. However, there are factors that client and management can determine in the pre-plan stage to minimize the unknown risk-factors. Realistic schedules, clear defect-removal procedures and transparent communication are all important factors to ensure software quality, software development quality and successful software projects.

9.3 Future research

This enhanced model is my contribution to improve software quality and the software development process. This model builds on the robust and time-tested model of Abdel-Hamid, and is thoroughly validated through my paper by both empirical data and technical tests. There are opportunities for future research to further improve my model. For example, the enhanced testing process includes several testing stages. However, what kind of testing is not discussed here. There is a variety of different testing schemes that include several techniques. Some testing-schemes are very complex and demanding, while others are more rudimentary and simpler to initiate. My analyses have been focused on the overall relationship between all efforts inside the model as a whole. For more complex analyses and advanced policy options, I would encourage future researchers to focus on isolated parts of the model in greater detail. Resource allocation for defect removal is also depending on types of schemes and the effort needed to deploy such efforts. This is not analysed in detail in this model. In addition the concept of errors has not been discussed here. In the original thesis Abdel-Hamid noted that errors are here described as either design or coding errors. This is done for model simplicity. However, in real-world software projects a wide variety of definitions on errors exists. It would be a useful contribution to develop the concept of errors in order to provide even more detailed policy-options for management. The more accurate information available provides bigger chance of success.

To obtain better software quality we must search for improved methods of software development and management. As observed by Dr. Deming; Quality must be built into the product or service from the start. It cannot be achieved through post-production control or maintenance.

Appendix 1: Equations

init	ActualFractionOfAManDayOnProject = INIT(NominalFractionOfAMDOnProject)

flow	ActualFractionOfAManDayOnProject = +dt*WorkRateAdjustmentRate

doc	ActualFractionOfAManDayOnProject = Actual Fraction of a Man-day on Project. The slack time is the fraction of project time lost to non-project activities - coffe-breaks, personal business. 100% increase is attainable because workers, in addition to partially compressing their slack time, may also work overtime hours. This will case actual productivity to be larger than potential productivity. The motivational effects of schedule pressure can push "Actual fraction of a man day on project to higher values (under postitive schedule pressure) or lower values (under negative schedule pressure).

init	AveActiveErrorDensityPerTask = ActiveErrorDensityPerTask

flow	AveActiveErrorDensityPerTask = +dt*ChngAveActiveErrorDensityPerTask

init	AveActivErrorDensityPerDSI = ActiveErrorDensityPerTask * 1000 / DSIPerTask

flow	AveActivErrorDensityPerDSI = +dt*ChngAveErrorDensityPerDSI

doc	AveActivErrorDensityPerDSI = The active error density will affect the generation of new errors through a delay. The reason is errors commited in one part of the system, will not affect other parts that are developed in parallell. Thus there is a delay until tasks in a later part of a sequence may be affected by earlier errors and thus generate new ones. (p.111)

init	ClarifiedReworkConsiderableTasks_afterClientReview = 0

flow	ClarifiedReworkConsiderableTasks_afterClientReview = +dt*ReworkConsiderableTasks_ClarificationRate

init	CumErrorsReworkedDuringTesting = 0

flow	CumErrorsReworkedDuringTesting = +dt*ActiveErrorsDetectionInTesting
	+dt*PassiveErrorsDetectionInTesting

doc	CumErrorsReworkedDuringTesting = The total number of errors rework. That means both passive and active errors.

init	CumlativeTrainingMD = 0

flow	CumlativeTrainingMD = +dt*DailyMDForTraining

doc	CumlativeTrainingMD = The cumulated number training mandays.

init	CumMDForTraining = 0

flow	CumMDForTraining = +dt*DailyMDForTraining

init	Cumulative_BadFixedErrors = 0

flow	Cumulative_BadFixedErrors = +dt*BadFixesGenerationRate

init	Cumulative_DetectedErrorsinTesting = 0

flow	Cumulative_DetectedErrorsinTesting = +dt*ActiveErrorsDetectionInTesting
	+dt*PassiveErrorsDetectionInTesting

init	Cumulative_EscapedErrors = 0

flow Cumulative_EscapedErrors = +dt*ErrorEscapeRate

doc Cumulative_EscapedErrors = The accumulation of errors that escaped.

init Cumulative_ManDayExpended = 0.0001

flow Cumulative_ManDayExpended = +dt*TotalDailyManpower

doc Cumulative_ManDayExpended = Cumulative mandays of all used in the project so far. (From manpower alloction sector)

init Cumulative_ReworkConsiderableErrors_afterClientReview = 0

flow Cumulative_ReworkConsiderableErrors_afterClientReview = +dt*ReworkConsiderableErrors_afterClientReview_TransferringRate

init Cumulative_TasksQAed = 0

flow Cumulative_TasksQAed = +dt*QACompletionRate

init CumulativeCodingDesignMD = 0

flow CumulativeCodingDesignMD = +dt*DailyManpowerForSoftwareDevelopment

doc CumulativeCodingDesignMD = (software development productivity)

init CumulativeDetectedErrors_Reworked = 0

flow CumulativeDetectedErrors_Reworked = +dt*DetectedErrors_ReworkRate

init CumulativeDetectedErrorsWaitingforRework = 0

flow CumulativeDetectedErrorsWaitingforRework = +dt*ErrorsDetectionRate

doc CumulativeDetectedErrorsWaitingforRework = The accumulation of errors detected.

init CumulativeDevelopmentMD = 0

flow CumulativeDevelopmentMD = +dt*DailyManpowerForDevelopmentAndTesting

doc CumulativeDevelopmentMD = The accumulation of mandays used for development and testing.

init CumulativeErrorsGeneratedDirectlyDuringWork = 0

flow CumulativeErrorsGeneratedDirectlyDuringWork = +dt*ErrorGenerationRate

init CumulativeNoReworkConsiderableDeviation = 0

flow CumulativeNoReworkConsiderableDeviation = +dt*NoReworkConsiderableDeviation_TransferringRate

init CumulativeNoTaggedErrorsObserved = 0

flow CumulativeNoTaggedErrorsObserved = +dt*PotentialDetectableEscapedErrors_IdentificationRate

init CumulativeQualityAssuranceMD = 0

flow CumulativeQualityAssuranceMD = +dt*DailyManpowerAllocatedForQA

doc CumulativeQualityAssuranceMD = Cumulative quality assurance mandays

init CumulativeReworkedErrors_Tagged = 0

flow CumulativeReworkedErrors_Tagged = +dt*ReworkedErrors_TagReleasingRate

init CumulativeReworkMD = 0

flow	CumulativeReworkMD = +dt*DailyManpowerAllocatedForRework

doc	CumulativeReworkMD = Cumulative rework mandays

init	CumulativeTaggedErrorsObserved = 0

flow	CumulativeTaggedErrorsObserved = +dt*TaggedErrors_IdentificationRate

init	CumulativeTasksDeveloped = 0

flow	CumulativeTasksDeveloped = +dt*SoftwareDevelopmentRate

doc	CumulativeTasksDeveloped = (3b-control subsector)

init	CumulativeTasksTested = 0

flow	CumulativeTasksTested = +dt*TestingRate

init	CumulativeTestingMD = 0

flow	CumulativeTestingMD = +dt*DailyManpowerForTesting

doc	CumulativeTestingMD = Cumulative Testing Man-days

(From system testing sector)

init	CurrentlyPerceivedJobSize = PercievedJobSizeI_DSI/DSIPerTask

flow	CurrentlyPerceivedJobSize = +dt*IncorporationOfDiscoveredTasksIintoProject

doc	CurrentlyPerceivedJobSize = This is the currently percieved job size. That is, the number of tasks that was underestimated initially will flow into this stock as it becomes visible and incorporated into the project. (From control subsystem)

init	DetectedDeviation_InTesting = 0

flow	DetectedDeviation_InTesting = -dt*DeviationReporting_Rate

+dt*PassiveErrorsDetectionInTesting

+dt*ActiveErrorsDetectionInTesting

init	DetectedErrors_WaitingforRework = 0

flow	DetectedErrors_WaitingforRework =

+dt*ReworkConsiderableErrors_afterClientReview_TransferringRate

+dt*TaggedErrors_IdentificationRate

-dt*DetectedErrors_ReworkRate

+dt*ErrorsDetectionRate

doc	DetectedErrors_WaitingforRework = The number of detected errors that is not yet fixed.

(From quality assurance and rework sector)

init	Exhaustion_level = 0

flow	Exhaustion_level = +dt*RateOfIncreaseinExhaustionLevel

-dt*RateOfDepletionIinExhaustionLevel

doc	Exhaustion_level = Exhaustion level.

Exhaustion is defined in the model due to overwork.

init	ExperiencedWorkforce = TeamSizeAtTheBeginningOfDesign

flow	ExperiencedWorkforce = -dt*ExperiencedTransferRate

```
            -dt*ExpEmplyeesQuitRate
            +dt*AssimilationRateOfNewEmployees
doc     ExperiencedWorkforce = People experience in the project. They are more productive than the
newly hired people.
init    Level_78 = 0
flow    Level_78 = +dt*ReworkConsiderableDeviation__SelectionRate
init    Level_79 = 0
flow    Level_79 = +dt*DeviationReporting_Rate
init    NewWorkforce = 0
flow    NewWorkforce = -dt*NewEmployeesTransferRateOut
            -dt*AssimilationRateOfNewEmployees
            +dt*HiringRate
doc     NewWorkforce = These are the persons that are recruted for working with the project. They
are less productive than the experienced workforce.
init    NonReworkConsiderableDeviations = 0
flow    NonReworkConsiderableDeviations =
+dt*NoReworkConsiderableDeviation_TransferringRate
            -dt*NoReworkConsiderableDeviations_ReportingRate
init    NoReworkConsiderableTasks_Reported = 0
flow    NoReworkConsiderableTasks_Reported = +dt*NoReworkConsiderableTasks_ReportingRate
            -dt*NoReworkConsiderableTasks_PresentationRate
init    NoReworkConsidrableTasks_afterClientReview = 0
flow    NoReworkConsidrableTasks_afterClientReview =
+dt*NoReworkConsidrableTasks_TransferingRate
init    PercentDevPerceivedCompleted = 0
flow    PercentDevPerceivedCompleted = +dt*ChangInPercentDevPerceivedCompleted
init    PercentTasksReportedComplete = 0
flow    PercentTasksReportedComplete = +dt*ChangInPercentTasksReportedComplete
doc     PercentTasksReportedComplete = This is the percent of tasks reported complete. This takes
time and is thus modelled as a smooth with a reporting delay of 10 days.
init    PercievedReworkManpowerNeededPerError = 0.5
flow    PercievedReworkManpowerNeededPerError = +dt*ChangeIn_P_RW_MP_N_P_Error
doc     PercievedReworkManpowerNeededPerError = This is a smoothed variable, because full and
imidiate action is seldom taken on a change of incoming information. There is a tendency to delay
action until the change is insistent. (From manpower allocation sector)
init    PercievedTestingProductivity = TestingProductivity
flow    PercievedTestingProductivity = +dt*AdjustmentOfPerceivedTestingProductivity
```

```
doc     PercievedTestingProductivity = "Percieved testing productivity" is a smooth because "full and
imidiate acition is seldom taken on a change of incoming information. There is a tendency to delay
action until the change is persistent. (p.123).(control subsector)
init    PlannedTestingSizeInMDBeforeWeStartTesting = TestingMD
flow    PlannedTestingSizeInMDBeforeWeStartTesting = +dt*AdjustmentOfTestingSizeMD
        +dt*IncreaseInTestingMDDueToDiscoveredTasks
init    PotentiallyDetectableErrors = 0
flow    PotentiallyDetectableErrors = +dt*PotentialDetectableEscapedErrors_IdentificationRate
        -dt*ErrorEscapeRate
        -dt*ErrorsDetectionRate
        +dt*ErrorGenerationRate
doc     PotentiallyDetectableErrors = Errors that are generated, and thus potentially may be detected.
(From quality assurance and rework subsector)
init    PresentedTasks_WaitingClientreview = 0
flow    PresentedTasks_WaitingClientreview = +dt*NoReworkConsiderableTasks_PresentationRate
        -dt*ReworkConsidrableTasks_SelectionRate
        -dt*NoReworkConsidrableTasks_TransferingRate
init    ReportedDeviation = 0
flow    ReportedDeviation = -dt*NoReworkConsiderableDeviation_TransferringRate
        -dt*ReworkConsiderableDeviation__SelectionRate
        +dt*DeviationReporting_Rate
init    ReworkConsiderableErrors_AfterSystemTesting = 0
flow    ReworkConsiderableErrors_AfterSystemTesting = -
dt*PotentialDetectableEscapedErrors_IdentificationRate
        -dt*TaggedErrors_IdentificationRate
        +dt*ReworkConsiderableDeviation__SelectionRate
init    ReworkConsiderableTasks_Reported = 0
flow    ReworkConsiderableTasks_Reported = -dt*ReworkConsiderableTasks_ClarificationRate
        +dt*ReworkConsiderableTasks_ReportingRate
init    ReworkConsidrableTasks_AfterClientReview = 0
flow    ReworkConsidrableTasks_AfterClientReview = -
dt*ReworkConsiderableTasks_ReportingRate
        +dt*ReworkConsidrableTasks_SelectionRate
init    ReworkedErrors = 0
flow    ReworkedErrors = -dt*ReworkedErrors_TagReleasingRate
        +dt*DetectedErrors_ReworkRate
doc     ReworkedErrors = (From quality assurance and rework sector)
```

```
init    ScheduledCompletionDate = TotalDevelopmentTime
flow    ScheduledCompletionDate = +dt*ScheduleAdjustment
doc     ScheduledCompletionDate = The "Scheduled completion date" represents not an actual date,
but the number of working days from the beginning of the project.
init    TasksDiscovered = 0
flow    TasksDiscovered = -dt*IncorporationOfDiscoveredTasksIintoProject
        +dt*TaskDiscoveryRate
init    TasksQAed = 0
flow    TasksQAed = +dt*QACompletionRate
        -dt*TestingRate
init    TasksWorked = 0
flow    TasksWorked = -dt*QACompletionRate
        +dt*SoftwareDevelopmentRate
doc     TasksWorked = Tasks that is developed but not yet quality assured (QAed)
init    TimeOfLastExhaustionBreakdown = -1
flow    TimeOfLastExhaustionBreakdown = +dt*BreakdownTimeSetter
doc     TimeOfLastExhaustionBreakdown = Time of Last Exhaustion Breakdown
init    TimeToRecover = 0
flow    TimeToRecover = +dt*ChangeInTimeToRecover
doc     TimeToRecover = Variable that controls time to de-exhaust
init    TotalJobSizeInMDs = DevelopmentMD+TestingMD
flow    TotalJobSizeInMDs = +dt*IncRInDevMDsDueToDiscoveredTasks
        +dt*IncreaseInTestingMDDueToDiscoveredTasks
        +dt*AdjustmentOfJobSizeMD
init    UndetectedActiveErrors = 0
flow    UndetectedActiveErrors = -dt*ActiveErrorsDetectionInTesting
        +dt*ActiveErrorRegenerationRate
        +dt*ActiveErrorGenerationRate
        -dt*ActiveErrorsRetiringRate
doc     UndetectedActiveErrors = The active existence of errors will produce more and more errors in
the system. For example, a design error will create errors in code manuals and so on, until detected.
All design errors are active, while coding of lower level modules in the final pahses of the project is
not.
init    UndetectedPassiveErrors = 0
flow    UndetectedPassiveErrors = -dt*PassiveErrorsDetectionInTesting
        +dt*PassiveErrorGenerationRate
        +dt*ActiveErrorsRetiringRate
```

doc UndetectedPassiveErrors = Some errors stay undetected, and are also passive. That means that they do not generate new errors. After a while they may be detected.
init UndiscoveredJobTasks = RealJobSizeTasks-CurrentlyPerceivedJobSize
flow UndiscoveredJobTasks = -dt*TaskDiscoveryRate
aux ActiveErrorGenerationRate = (ErrorEscapeRate+BadFixesGenerationRate)*FracOfEscapedErrorsActive
doc ActiveErrorGenerationRate = The rate at which new and active errors are generated.
aux ActiveErrorRegenerationRate = SoftwareDevelopmentRate * AveActiveErrorDensityPerTask * MultiplierToActiveErrorRegenDueToErrorDensity
aux ActiveErrorsDetectionInTesting = MIN(TestingRate*ActiveErrorDensityPerTask, UndetectedActiveErrors/TIMESTEP)
aux ActiveErrorsRetiringRate = UndetectedActiveErrors*ActiveErrorsRetiringFraction
doc ActiveErrorsRetiringRate = When undetected active errors cease to reproduce, they effectively become "Undetected passive errors". The active error retirement rate is is regulated through the fraction of active errors that become passive every unit of time.
aux AdjustmentOfJobSizeMD = (MDReportedStillNeeded+Cumulative_ManDayExpended-TotalJobSizeInMDs)/DelayInAdjJobSizeMD
doc AdjustmentOfJobSizeMD = The "Rate of adjusting the job size in man days" is the rate at which the "Total_Job Size_in_ManDays" is adjusted upward or downward to what is percieved as its new value. (p.124) The goal itself is given by "Reported_shortage_in_man_days_still_needed+Cumulative_mandays_expended" (p.124), and the adjsutment to this goal is not instantaneous one. This adjustment process is the projects LAST reaction to manday shortages or excesses. (p.124)
aux AdjustmentOfPerceivedTestingProductivity = (TestingProductivity - PercievedTestingProductivity)/TimeToAdjustPerceivedTestProd
doc AdjustmentOfPerceivedTestingProductivity = IF(((Testing_productivity - Percieved_testing_productivity)/Time_to_Smooth_Test_prod + Percieved_testing_productivity) > 1, (Testing_productivity - Percieved_testing_productivity)/Time_to_Smooth_Test_prod, 1)

aux AdjustmentOfTestingSizeMD = (IF(FracOfEffortForSystemTesting>=0.9, 1, 0)) * AdjustmentOfJobSizeMD
doc AdjustmentOfTestingSizeMD = If the project is in the testing phase, i.e. "Frac_of_Effort_for_System_Testing" >= 90%, then any changes in the rate of adjusting the jobsize in mandays will be from systems testing man days. This is because other actvities (development: design/coding) are almost finished.
aux AssimilationRateOfNewEmployees = NewWorkforce/AverageAssimilationDelay

doc	AssimilationRateOfNewEmployees = Time rate at which people go from being new to becoming experienced in the project.
aux	BadFixesGenerationRate = (DetectedErrors_ReworkRate*FractionofBadFixes)*MutipltoBadFixGenDueto_ReworkedErrorDensity
doc	BadFixesGenerationRate = (FixedErrors_TagReleasingRate*FractionofBadFixes)*1
aux	BreakdownTimeSetter = (MAX(TimeOfLastExhaustionBreakdown, BreakdownIndicator) - (TimeOfLastExhaustionBreakdown))/TIMESTEP
doc	BreakdownTimeSetter = MAX(Time_of_Last_Exhaustion_Breakdown, Breakdown) - (Time_of_Last_Exhaustion_Breakdown/TIMESTEP)
aux	ChangeIn_P_RW_MP_N_P_Error = (ReworkManpowerNeededPerAverageError-PercievedReworkManpowerNeededPerError)/TimeToAdjust_P_RW_MP_N_P_Error
doc	ChangeIn_P_RW_MP_N_P_Error = The rate at which the perception about the rework manpower neede per error changes.
aux	ChangeInTimeToRecover = IF(Exhaustion_level/MaximumTolerableExhaustion >= 0.1,1,-TimeToRecover/TIMESTEP)
doc	ChangeInTimeToRecover = The longer the exhaustion level is larger than 10% of the Max Exhaustion level, then more time is added to the time needed to recover. Once the Exhaustion Level has reached a value of less than 10% of the maximum Exhaustion level, then the time to recover is set to 0.
aux	ChangInPercentDevPerceivedCompleted = (MAX((100-((MDReportedStillNeeded-MDPercievedStillNededForTesting)/(TotalJobSizeInMDs - PlannedTestingSizeInMDBeforeWeStartTesting))*100), PercentDevPerceivedCompleted) - PercentDevPerceivedCompleted) / ReportingDelay
doc	ChangInPercentDevPerceivedCompleted = This is the rate for smoothing the information about "the percent of development percieved complete".
aux	ChangInPercentTasksReportedComplete = ((100 - (MDReportedStillNeeded/TotalJobSizeInMDs)*100) - PercentTasksReportedComplete) / ReportingDelay
doc	ChangInPercentTasksReportedComplete = This is the smoothing rate of the "Percent_Tasks_Reported_Complete".
aux	ChngAveActiveErrorDensityPerTask = (ActiveErrorDensityPerTask-AveActiveErrorDensityPerTask)/TimeToSmoothActiveErrorDensity
doc	ChngAveActiveErrorDensityPerTask = First order information smooth of the active error density.
aux	ChngAveErrorDensityPerDSI = ((ActiveErrorDensityPerTask*1000/DSIPerTask) - AveActivErrorDensityPerDSI) / TimeToSmoothActiveErrorDensity

aux DailyManpowerAllocatedForQA = MIN(ActualFractionOfManpowerForQA*TotalDailyManpower,0.9*DailyManpowerAvailiableAfterTraining)

doc DailyManpowerAllocatedForQA = Daily manpower allocated for quality assurance. Daily manpower for QA is 15% of total daily manpower, unless daily manpower after training is low because of the amount of new employees. This requires more quality assurance relative to production. I.e 90% of available manpower is allocated for QA. (From manpower allocation sector)

aux DailyManpowerAllocatedForRework = MIN(DesiredErrorCorrectionRate*PercievedReworkManpowerNeededPerError,DailyManpowerForSoftwareProduction)

doc DailyManpowerAllocatedForRework = Daily manpower allocation for rework is the desired error correction rate times the Percieved_rework_Man power_needed_per_error, unless it is limited by the Daily_manpower_for_software_production. That is - the availiable manpower. (Manpower alloction sector)

aux DailyManpowerForDevelopmentAndTesting = DailyManpowerForSoftwareProduction-DailyManpowerAllocatedForRework

doc DailyManpowerForDevelopmentAndTesting = Daily manpower for development and testing is the manpower for software production minus manpower used for rework.
(From manpower allocation sector)

aux DailyManpowerForSoftwareDevelopment = DailyManpowerForDevelopmentAndTesting*(1-FracOfEffortForSystemTesting)

doc DailyManpowerForSoftwareDevelopment = Manpower used for software development per day.

aux DailyManpowerForTesting = DailyManpowerForDevelopmentAndTesting*FracOfEffortForSystemTesting

doc DailyManpowerForTesting = Daily Manpower for Testing

aux DailyMDForTraining = NewWorkforce*TrainersPerNewEmployee

doc DailyMDForTraining = The use of workforce to train newly hired workforce. (From the human resource management subsystem.)

aux DetectedErrors_ReworkRate = MIN(PotentialReworkRate, DetectedErrors_WaitingforRework/TIMESTEP)

doc DetectedErrors_ReworkRate = MIN(PotentialReworkRate, DetectedErrorsWaitingforRework/TIMESTEP) The Rework Rate is a function of how much effort is allocated to rework activities and the rework manpower needed per error.
(From quality assurance and reworksector)

aux DeviationReporting_Rate = DetectedDeviation_InTesting/DeviationReportPrepration_Time

aux ErrorEscapeRate = (QACompletionRate*AverageNumberOfErrorsPerTask)

doc ErrorEscapeRate = The rate at which errors escape detection.
(From quality assurance and rework sector)

aux ErrorGenerationRate = SoftwareDevelopmentRate*ErrorsPerTask

doc ErrorGenerationRate = The rate at which errors are generated during development. (from QA and rework subsector)

aux ErrorsDetectionRate =
MIN(PotentialErrorDetectionRate,PotentiallyDetectableErrors/TIMESTEP)

doc ErrorsDetectionRate = The rate at which potentially detectable errors are detected.

aux ExpEmplyeesQuitRate = ExperiencedWorkforce/Average_employment_time

doc ExpEmplyeesQuitRate = The rate at which experienced workforce is leaving the project. This is a way of capturing turover.

aux ExperiencedTransferRate = MIN(ExperiencedWorkforce/TIMESTEP, TransferRate - NewEmployeesTransferRateOut)

doc ExperiencedTransferRate = If there is a desire to have less pople in the project and there is few experienced emplyees, experienced employees will have to leave. However not more than those avialiable in the stock.

aux HiringRate = MAX(0,WorkforceGap/HiringDelay)

doc HiringRate = The rate at which new people are recruited to the project.

aux IncorporationOfDiscoveredTasksIintoProject = MAX(DELAYMTR(TaskDiscoveryRate, AveDelayInIncorporatingDiscoveredTasks,3),0)

doc IncorporationOfDiscoveredTasksIintoProject = To process of incorporating additional tasks that are discovered, is a process that takes time. Thus, it is modelled as a third order delay.

aux IncreaseInTestingMDDueToDiscoveredTasks =
(IncorporationOfDiscoveredTasksIintoProject/PercievedTestingProductivity)*FractOfAddnlTasksAddingToMDs

doc IncreaseInTestingMDDueToDiscoveredTasks = If there is a decision to incorporate tasks discovered, into the projects man-days estimate, such an adjustment involves producing two estimates,one for the effort to develop and QA new tasks and the other for system testing. Any such adjustment will trigger further adjstments in either the project's schduled completion date, the workforce level or both. This is done in the planning sector. (p128)

aux IncRInDevMDsDueToDiscoveredTasks =
(IncorporationOfDiscoveredTasksIintoProject/AssumedDevelopmentInclQAProductivity)*FractOfAddnlTasksAddingToMDs

doc IncRInDevMDsDueToDiscoveredTasks = This rate is an adjustment process due to underestimation of tasks.

aux NewEmployeesTransferRateOut = MIN(TransferRate, NewWorkforce/TIMESTEP)

doc NewEmployeesTransferRateOut = If there is a need to reduce the number of hired people, the newly hired will be the first to leave, but not more than those availiable.
aux NoReworkConsiderableDeviation_TransferringRate = (MIN((PotentialClassification_rate*ReportedErrorsperTask),ReportedDeviation/TIMESTEP))*(1-FractionofConsiderableErrors_DuetoErrorsType)*Mutiplierto_NoReworkConsiderableErrors_DuetoSchedulePressure
doc NoReworkConsiderableDeviation_TransferringRate = (MIN((PotentialClassification_rate*Auxiliary_177),ReportedErrors_WaitingClassification/TIMESTEP))*(1-MultiplierTo_ReworkConsiderableErrors_DuetoErrorType)*(1-MultiplierTo_ReworkConsiderableError_DuetoSchedulePressure)
aux NoReworkConsiderableDeviations_ReportingRate = NonReworkConsiderableDeviations/NoConsiderableDeviation_ReportPrepration_Time
aux NoReworkConsiderableTasks_PresentationRate = (MIN(PotentialPresentation_Rate,NoReworkConsiderableTasks_Reported/TIMESTEP))
aux NoReworkConsiderableTasks_ReportingRate = NoReworkConsiderableDeviations_ReportingRate/(ReportedDeviations_perTask+0.0001)
aux NoReworkConsidrableTasks_TransferingRate = (MIN(PotentialClientReviewing_rate,PresentedTasks_WaitingClientreview/TIMESTEP))*(1-FractionofConsiderabletasks_DuetoErrorsType)
doc NoReworkConsidrableTasks_TransferingRate = (MIN(PotentialClientTesting_rate,PresentedTasks_WaitingClientTest/TIMESTEP))*(1-MultiplierTo_ReworkConsiderableTasks_DuetoErrorType)
aux PassiveErrorGenerationRate = (ErrorEscapeRate+BadFixesGenerationRate)*(1-FracOfEscapedErrorsActive)
doc PassiveErrorGenerationRate = This is the rate of the generation of passive errors. That means the fraction of errors generated that is not active: thus: 1- Frac_of_Escaping_Errors_Active.
aux PassiveErrorsDetectionInTesting = MIN(TestingRate*PassiveErrorDensityPerTask, UndetectedPassiveErrors/TIMESTEP)
doc PassiveErrorsDetectionInTesting = This is that rate at which passive errors are detected and corrected.
aux PotentialDetectableEscapedErrors_IdentificationRate = (MIN((PotentialIdentification_Rate*Fractionof_NoTaggedErrors_perTask),ReworkConsiderableErrors_AfterSystemTesting/TIMESTEP))
doc PotentialDetectableEscapedErrors_IdentificationRate = (MIN(PotentialTagReleasing_Rate,ReworkConsiderableErrors_WaitingTagReleasing/TIMESTEP))*Fractionf_NoLabledErrors

aux QACompletionRate = MAX(DELAYMTR(SoftwareDevelopmentRate, AverageQATime,3,0),0)
doc QACompletionRate = The QA rate has a non-characteristic formulation;a third order delay. The QA rate is independent of the QA effort and its productivity. QA effort or 'window' is planned allocated and allocated, usually in a fixed schedule of group fucntions. Therefore no backlogs develop. (p102-103) (From quality assurance and rework sector)
aux RateOfDepletionIinExhaustionLevel = IF(RateOfIncreaseinExhaustionLevel <= 0, Exhaustion_level/ExhaustionDepletionTime,0)
doc RateOfDepletionIinExhaustionLevel = Rate of Depletion in Exhaustion Level
Once a period of overwork ends, either because of threshold or cease in schedule pressure the workforce returns to normal work rate. The rate is modeled as first order exponential delay, with a time equal to 4 weeks.
aux RateOfIncreaseinExhaustionLevel = GRAPH((1-ActualFractionOfAManDayOnProject)/(1-NominalFractionOfAMDOnProject),-0.5,0.1,[2.5,2.2,1.9,1.6,1.3,1.15,0.9,0.8,0.7,0.6,0.5,0.4,0.3,0.2,0,0"Min:0;Max:2.5;Zoom"])
doc RateOfIncreaseinExhaustionLevel = Rate of Increas in Exhaustion Level.
When actual fraction of mandays on project is the same as or less than normal there is no increase in exhaustion level.
Exhaustion rate is a function of the slack, (1-AFMDPJ (Actual Fraction of a ManDay on Project)), relative to the Average Slack (1-NFMDPJ (Nominal Fraction of a Man-Day on Project)), - a constant. Thus, the exhaustion rate of work force is a function of the compression in the average slack time and eventually work overtime (1-AFMDPJ) <0 (negative slack).
When 1-AFMDPJ approches zero so that the slack is absorbed and then moves negative, people would be not only be compressing their slack time but also working overtime. (p.89)
aux ReworkConsiderableDeviation__SelectionRate = (MIN((PotentialClassification_rate*ReportedErrorsperTask),ReportedDeviation/TIMESTEP))*FractionofConsiderableErrors_DuetoErrorsType
doc ReworkConsiderableDeviation__SelectionRate = (MIN((PotentialClassification_rate*Auxiliary_177),ReportedErrors_WaitingClassification/TIMESTEP))*(MultiplierTo_ReworkConsiderableErrors_DuetoErrorType)*(MultiplierTo_ReworkConsiderableError_DuetoSchedulePressure)
aux ReworkConsiderableErrors_afterClientReview_TransferringRate = ReworkConsiderableTasks_ClarificationRate*ReportedDeviations_perTask
aux ReworkConsiderableTasks_ClarificationRate = (MIN(PotentialIClarification_Rate,ReworkConsiderableTasks_Reported/TIMESTEP))

aux	ReworkConsiderableTasks_ReportingRate = ReworkConsidrableTasks_AfterClientReview/ReworkConsiderableTask_ReportingTime
aux	ReworkConsidrableTasks_SelectionRate = (MIN(PotentialClientReviewing_rate,PresentedTasks_WaitingClientreview/TIMESTEP))*(Fractionof Considerabletasks_DuetoErrorsType)
aux	ReworkedErrors_TagReleasingRate = MIN(PotentialTagReleasing_Rate, ReworkedErrors/TIMESTEP)
aux	ScheduleAdjustment = (IndicatedCompletionDate - ScheduledCompletionDate)/ScheduleAdjustmentTime
aux	SoftwareDevelopmentRate = MIN(DailyManpowerForSoftwareDevelopment*DevelopmentProductivity,NewTasksPercievedRemaining/TIMESTEP)
doc	SoftwareDevelopmentRate = Software development rate is a function of (p77):
1. tasks per day
2. software developed of tasks developed
3. software development productivity of tasks per man-day.			(From software development productivity subsector)
aux	TaggedErrors_IdentificationRate = (MIN((PotentialIdentification_Rate*Fractionof_TaggedErrors_perTask),ReworkConsiderableErrors_AfterSystemTesting/TIMESTEP))
doc	TaggedErrors_IdentificationRate = (MIN(PotentialTagReleasing_Rate,ReworkConsiderableErrors_WaitingTagReleasing/TIMESTEP))*Fractionof_LabledErrors
aux	TaskDiscoveryRate = MAX(UndiscoveredJobTasks*PercentOfUndiscoveredTasksDiscoveredPerDay/100,0)
doc	TaskDiscoveryRate = The rate at which undiscovered tasks are discovered tends to increase as the project develops, because the teams level of knowledge of what the software product is intended to do increases. (p.126) Thus, the "Percent_of_Undiscovered_Tasks_Discovered_per_Day" is a variable not a constant.
aux	TestingRate = MIN(TasksQAed/TIMESTEP, DailyManpowerForTesting/TestingMapowerNeededPerTask)
doc	TestingRate = The rate at which developed (designed and coded) tasks are tested, is determined by dividing the "Daily manpower for Testing" by the "Testing manpower needed per task". If there is five man-days allocated and it takes one manday to test a task, five tasks will be tested per day. (p. 115)
aux	TotalDailyManpower = TotalWorkforceLevel*AverageDailyManpowerPerStaff

doc	TotalDailyManpower = Total daily manpower. The daily manpower in man-days availiable per day.
aux	WorkRateAdjustmentRate = (WorkrateSought-ActualFractionOfAManDayOnProject)/WorkRateAdjustmentDelay
doc	WorkRateAdjustmentRate = The rate at which the "Actual_fraction_of_a_man_day_on_project" is adjusted. It is adjusted towards its goal: "Workrate_sought" with a delay equal to "Work_rate_adjustment_delay".
aux	ActiveErrorDensityPerTask = UndetectedActiveErrors/(TasksQAed + 0.1)
aux	ActiveErrorsRetiringFraction = GRAPH(PercentOfJobActuallyWorked,0,0.1,[0,0,0,0,0.01,0.02,0.03,0.04,0.1,0.3,1"Min:0;Max:1"])
doc	ActiveErrorsRetiringFraction = The fraction is a function of the development phase. In the coding phase the fraction is zero (as design errors have to be coded to generate new errors). As the project approaches toward the last stages of development - the coding of the lower level functional modules - error propagation quickly decreases, and the "Retirement fraction" consequently increases sharply and reaches a value of 1 at the end of development. (p.114)
aux	ActualDailyManPower_forClarification = MIN(MaxDailyManPower_forClarification,DesiredDailyManPower_forClarification)
aux	ActualDailyManPower_forClassification = MIN(MaxDailyManPower_forClassification,DesiredDailyManPower_forClassification)
aux	ActualDailyManPower_forClientReview = MIN(MaxDailyManpower_forClientReview,DesiredDailyManPower_forClientReview)
doc	ActualDailyManPower_forClientReview = MIN(DailyClientManpower_forClientTest,DesiredDailyManPower_forClientTest)
aux	ActualDailyManPower_forIdentificationn = MIN(MaxDailyManPower_forIdentificationn,DesiredDailyManPower_forIdentificationn)
aux	ActualDailyManPower_forPresentation = MIN(MaxDailyManPower_forPresentation,DesiredDailyManPower_forPresentation)
aux	ActualDailyManPower_forTagging = MIN(MaxDailyManPower_forTagging,DesiredDailyManPower_forTagging)
aux	ActualFractionOfManpowerForQA = PlannedFractionOfManpowerForQA*(1+PercentAdjustmentInPlannedFractionOfManpowerForQA)
doc	ActualFractionOfManpowerForQA = Actual fraction of manpower for quality assurance is determined by how the schedule pressure affects the planned fraction of manpower for quality assurance. Under pressure, the quality drops.
aux	ActualTestingProductivity = (CumulativeTasksTested)/(CumulativeTestingMD+0.001)
doc	ActualTestingProductivity = (CumulativeTasksTested+CumulativeTasksClientTested)/(CumulativeTestingMD+0.001)

The "Actual testing productivity" is determined by dividing "Cumulative_tasks_tested" by "Cumulative_Testing_Man_Days".

aux AllErrors = ReworkedErrors + CumErrorsReworkedDuringTesting + DetectedErrors_WaitingforRework + PotentiallyDetectableErrors + UndetectedActiveErrors + UndetectedPassiveErrors

doc AllErrors = All Errors

aux AllErrorsDetectedAndRemaining = UndetectedActiveErrors+UndetectedPassiveErrors+CumErrorsReworkedDuringTesting

doc AllErrorsDetectedAndRemaining = All Errors that Escaped and were Generated

aux AllErrorsReworkedDuringDevelopmentAndTesting = ReworkedErrors+CumErrorsReworkedDuringTesting

doc AllErrorsReworkedDuringDevelopmentAndTesting = All Errors Reworked in Development and Testing

aux AssumedDesignAndCodingProductivity = ProjectedDesignAndCodingProductivity*WeightOnProjectedProductivity + PerceivedDesignAndCodingProductivity*(1-WeightOnProjectedProductivity)

aux AssumedDevelopmentInclQAProductivity = ProjectedDevelopmentInclQAProductivity*WeightOnProjectedProductivity + PercievedDevelopmentInclQAProductivity*(1-WeightOnProjectedProductivity)

doc AssumedDevelopmentInclQAProductivity = Assumed development productivity is a weighted average of percieved and projected development productivity, where the "Weight to project prouctivity" moves from 1 at the beginning of the project, to zero at the end of the development phase (design + coding). Thus giving more wheight to percived prod, and less to projected prod as the project developes. (From control subsystem)

aux AverageNominalPotentialProduvtivity = FractExperWorkforce*NominalPotentialProductivityOfExpEmployee+(1-FractExperWorkforce)*NominalPotentialProductivityOfNewEmployee

doc AverageNominalPotentialProduvtivity = The average nominal potential productivity, given the mixture of experienced and new emplyees. This is a variable dependent on the mixture.

aux AverageNumberOfErrorsPerTask = MAX(PotentiallyDetectableErrors/(TasksWorked+0.0001), 0)

doc AverageNumberOfErrorsPerTask = The average number of errors per tasks.

aux Benefit_Cost = CumulativeDetectedErrors_Reworked/(Cumulative_ManDayExpended+0.0001)

aux BreakdownIndicator = IF(OverworkDurationThreshold=0,TIME+TIMESTEP,0)

aux CeilingOnNewHires = FullTimeEquivalentExperiencedWorkforce*MaxNewHireesPerFulltimeExperiencedStaff

doc CeilingOnNewHires = The rate of hiring that of new project members that project management feels its fully integrated staff can handle. This is an implicit policy.
aux CeilingOnTotalWorkforce = ExperiencedWorkforce+CeilingOnNewHires
doc CeilingOnTotalWorkforce = The maximum number of employees management wish to hire.
aux ClarificationManPowerNeeded_perTask = TestingMapowerNeededPerTask
aux ClassificationManPoweNeeded_perError = TestingManpowerNeeded_PerAverageError*PercentClassificationManpower_PerError
aux ClassificationManPoweNeeded_perTask = TestingMapowerNeededPerTask*PercentClassificationManpower_PerTask
doc ClassificationManPoweNeeded_perTask = TestingManpowerNeeded_PerAverageError*PercentClassificationManpower_PerAverageError
Armin: (1/(Manpower_Productivity*ErrorsPerTask))
aux ClientManpowerNeeded_PerTask = (1/ClientManpower_Productivity)
doc ClientManpowerNeeded_PerTask = IF(Presented_NoReworkConsiderableDeviation_Density>0, 1/(ClientProductivity*Presented_NoReworkConsiderableDeviation_Density), 0.00001)
Check Formula!!! 1/(ClientProductivity*(Presented_NoReworkConsiderableDeviation_Density+0))
aux CodingDesignTrainingMD = CumMDForTraining+CumulativeCodingDesignMD
aux CommunicationOverhead = GRAPH(TotalWorkforceLevel,0,5,[0,0.015,0.06,0.135,0.24,0.375,0.54"Min:0;Max:1"])
doc CommunicationOverhead = It is the average team member's drop in productivity below his nominal productivity as a result of team comunication, where communicatoin includes verbal communication, documantation and any additional work, such as that due to interfaces. (p93) Overhead increases with a larger workforce.
aux Cost_Benefit = Cumulative_ManDayExpended/(CumulativeDetectedErrors_Reworked+0.0001)
aux DailyManpowerAvailiableAfterTraining = TotalDailyManpower-DailyMDForTraining
doc DailyManpowerAvailiableAfterTraining = The daily manpower avaliable after training overhead. The total manpower minus manpower used for training new employees in the project.
aux DailyManpowerForSoftwareProduction = DailyManpowerAvailiableAfterTraining-DailyManpowerAllocatedForQA
doc DailyManpowerForSoftwareProduction = The daily manpower for software production is the total manpower minus manpower used for training new employees, and manpower allocated for quality assurance work.
aux DelayInAdjJobSizeMD = GRAPH(TimeRemaining,0,20,[0.5,3"Min:0;Max:3"])

doc DelayInAdjJobSizeMD = This delay is normaly 3 days, but as the project approaches its final stages it drops to 0.5 as the last tasks will have to be finished and thus the JS is obvious and one does not need to see any persistent change at this stage. It is simply to have the project finished.

aux DesiredDailyManPower_forClarification = (ReworkConsiderableTasks_Reported*ClarificationManPowerNeeded_perTask)/DesiredClarification_Delay

doc DesiredDailyManPower_forClarification = Armin: Unit!!! in this formula it becomes just MD!

aux DesiredDailyManPower_forClassification = (ReportedDeviation*ClassificationManPoweNeeded_perError)/DesiredClassification_Time

doc DesiredDailyManPower_forClassification = Armin: Unit!!!

aux DesiredDailyManPower_forClientReview = (PresentedTasks_WaitingClientreview*PotentialClientManpowerNeededtoReview_PerTask)/DesiredClientReview_Delay

doc DesiredDailyManPower_forClientReview = Armin: Unit!!! in this formula it becomes just MD!

aux DesiredDailyManPower_forIdentificationn = (ReworkConsiderableErrors_AfterSystemTesting*IdentificationManPoweNeeded_PerError)/DesiredIdentificationn_Time

doc DesiredDailyManPower_forIdentificationn = Armin: Unit!!! in this formula it becomes just MD!

aux DesiredDailyManPower_forPresentation = (NoReworkConsiderableTasks_Reported*PresentationManPoweNeeded_PerTask)/DesiredPresentation_Time

doc DesiredDailyManPower_forPresentation = Armin: Unit!!! in this formula it becomes just MD!

aux DesiredDailyManPower_forTagging = (ReworkedErrors*TaggingManPoweNeeded_PerError)/DesiredTagging_Time

doc DesiredDailyManPower_forTagging = Armin: Unit!!! in this formula it becomes just MD!

aux DesiredErrorCorrectionRate = DetectedErrors_WaitingforRework/DesiredReworkDelay

doc DesiredErrorCorrectionRate = Desired error correction rate is the rate at which the discovered errors are corrected given the desired rework delay.

aux DetectedErrorDensityDSI = DetectedErrorsDensity*1000/DSIPerTask

aux DetectedErrorsDensity = MAX(DetectedErrors_WaitingforRework/(TasksQAed+0.0001), 0)

aux DevelopmentMD = TotalMD*FractOfEffortAssumedNeededForDevelopment

doc DevelopmentMD = 80% of the resurces are estimated to be consumed in the development phase. (From initialisation sector) aux DevelopmentProductivity = PotentialDevelopmentProductivity*MultiplierToProductivityDueToMotivationAndCommunicationLosses

doc DevelopmentProductivity = Psycholigical model of group productivity. Losses due to faulty process refer basically to communication and motivation losses. Potential productivity is the maximum productivity that can be obtained if there are no faulty processes.

aux EffectOfExhaustionOnOverWorkDurationThreshold = GRAPH(Exhaustion_level/MaximumTolerableExhaustion,0,0.1,[1,0.9,0.8,0.7,0.6,0.5,0.4,0.3,0.2,0.1,0"Min:0;Max:1"])

doc EffectOfExhaustionOnOverWorkDurationThreshold = Effect of Exhaustion on Overwork Duration Threshold

With increased exhaustion the overwork duration threshold is reduced. When Exhaustion level reaches the maximum tolerable level of exhaustion the overwork duration threshold drives to zero.

aux EffectOfProgressOnWeightOnProjectedProductivity = GRAPH(PercentOfJobPerceivedWorked/100,0,0.1,[1,1,1,1,1,1,0.975,0.9,0.75,0.5,0"Min:0;Max:1"])

doc EffectOfProgressOnWeightOnProjectedProductivity = Moves from one at the beginning of the project and ends up with zero when all tasks are develped. (p.121)

aux EffectOfWorkrateSoughOn_WRAD = IF(WorkrateSought>=ActualFractionOfAManDayOnProject,1,0.75)

aux EffOfResourceExpenditureProjectedProdWeight = GRAPH((1-MDPercievedRemainingForNewTasks/(TotalJobSizeInMDs - PlannedTestingSizeInMDBeforeWeStartTesting)),0,0.1,[1,1,1,1,1,1,0.975,0.9,0.75,0.5,0"Min:0;Max:1"])

doc EffOfResourceExpenditureProjectedProdWeight = Moves from one at the beginning of the project and ends up with zero when all estimated development resouces are expended (p.121)

aux ErrorDensityDSI = AverageNumberOfErrorsPerTask*1000/DSIPerTask

doc ErrorDensityDSI = The error density in errors per KDSI.

aux ErrorsPerTask = NominalNoOfErrorsCommittedPerTask*MultiplToErrGenerationDueToSchedulePressure*MulltiplToErrGenDueToWorkforceMix

doc ErrorsPerTask = Total number of errors commited per task, including effects from workforce mix and schedule pressure.

aux ErrorsReworkedDuringDevelopmen_perTask = MAX(CumulativeReworkedErrors_Tagged/(Cumulative_TasksQAed+0.0001), 0)

aux EstTimeRecovered = TimeOfLastExhaustionBreakdown+TimeToRecover

aux FracOfEffortForSystemTesting =
GRAPH(NewTasksPercievedRemaining/CurrentlyPerceivedJobSize,0,0.04,[1,0.5,0.28,0.15,0.05,0"Min:0;Max:1"])
doc FracOfEffortForSystemTesting =
RT:GRAPH(NewTasksPercievedRemaining/CurrentlyPerceivedJobSize,0,0.04,[1,0.2,0,0,0,0"Min:0;Max:1"]) RA1:
GRAPH(NewTasksPercievedRemaining/CurrentlyPerceivedJobSize,0,0.1,[1,0.58,0.31,0.14,0.05,0,0,0,0,0,0"Min:0;Max:1"])
RA2:GRAPH(NewTasksPercievedRemaining/CurrentlyPerceivedJobSize,0,0.1,[1,0.65,0.45,0.3,0.21,0.14,0.09,0.04,0,0,0"Min:0;Max:1"])
N:GRAPH(NewTasksPercievedRemaining/CurrentlyPerceivedJobSize,0,0.04,[1,0.5,0.28,0.15,0.05,0"Min:0;Max:1"]) Fraction of Effort for System Testing. This variable represents the switch in manpower allocation from development to testing. The value for this variable is initially zero since no effort is initially allocated for System Testing. When development (desing and coding) is percieved to be completed, this value becomes 1 (i.e. 100%) with a certain overlap. (p.115 and p.79) (From software development productivity subsector)
aux FracOfEscapedErrorsActive =
GRAPH(PercentOfJobActuallyWorked,0,0.1,[1,1,1,1,0.95,0.85,0.5,0.2,0.075,0,0"Min:0;Max:1"])
doc FracOfEscapedErrorsActive = Fraction Active Errors, Fraction of escaping errors that will be active, or (% active errors), shows how the mix of active and passive errors as the project progresses. There is only active errors in the beginning, but in the coding phase there are more and more passive errors, such as coding errors in lower level modules.
aux FractExperWorkforce = ExperiencedWorkforce/TotalWorkforceLevel
doc FractExperWorkforce = The fraction of the total workforce that is experienced. (From human resource management subsystem)
aux Fractionof_NoTaggedErrors_perTask =
MAX((Cumulative_EscapedErrors/(Cumulative_TasksQAed+0.0001)), 0)
doc Fractionof_NoTaggedErrors_perTask =
MAX(CumulativeEscapedErrors/(CumulativeTasksQAed+0.0001), 0)
100*CumulativeEscapedErrors/(CumQA+0.001)
aux Fractionof_TaggedErrors_perTask =
MAX((Cumulative_BadFixedErrors/(Cumulative_TasksQAed+0.0001)), 0)
doc Fractionof_TaggedErrors_perTask =
MAX(Cumulative_BadFixedErrors/(CumulativeTasksQAed+0.0001), 0)
100*Cumulative_BadFixedErrors/(CumQA+0.001)

aux FractionofBadFixes =
MultiplToBadFixesGenDueTo_WorkforceMix*MutipltoBadFixGenDueto_DetectedErrorDensity*MultiplToBdFixesGenDueTo_SchedulePressure*MultiplToBdFixesGenDueTo_ErrorType
doc FractionofBadFixes =
MultiplToBadFixesGenDueToWorkforceMix*MutipltoBadFixGenDuetoDetectedErrorDensity*MultiplToBdFixesGenDueToSchedulePressure*NominalNumberOfBadFixesCommittedPerTask
Armin: *NomNoOfBadFixesCommitedPerTask *MultiplToBadFixesGenDueToErrorType*
aux FractionofConsiderableErrors_DuetoErrorsType =
GRAPH(PercentOfJobActuallyWorked,0,0.1,[1,1,1,1,1,0.98,0.92,0.79,0.1,0.03,0"Min:0;Max:1"])
doc FractionofConsiderableErrors_DuetoErrorsType =
GRAPH(PercentOfJobActuallyWorked,0,0.1,[1,1,1,1,1,0.98,0.92,0.8,0.13,0.02,0"Min:0;Max:1"])
GRAPH(PercentOfJobActuallyWorked,0,0.1,[1,1,1,1,1,1,1,1,0.21,0.01,0"Min:0;Max:1"])
aux FractionofConsiderabletasks_DuetoErrorsType =
GRAPH(PercentOfJobActuallyWorked,0,0.1,[0.6,0.594,0.58,0.552,0.5,0.432,0.333,0.227,0.111,0.04,0"Min:0;Max:0.6"])
doc FractionofConsiderabletasks_DuetoErrorsType =
GRAPH(PercentOfJobActuallyWorked,0,0.1,[0.6,0.577,0.541,0.492,0.41,0.3,0.099,0.045,0.017,0.006,0"Min:0;Max:0.6"])
aux FractOfAddnlTasksAddingToMDs =
GRAPH(RelativeSizeOfDiscoveredTasks/(MaxRelSizeOfAdditionsToleratedWithoutAddingToProjectMD + 0.001),0,0.2,[0,0,0,0,0,0,0.7,0.9,0.975,1,1"Min:0;Max:1"])
doc FractOfAddnlTasksAddingToMDs = Once the relative size is determined, it is then compared with the threshold value; The "Maximum relative Size of Additions Tolerated Without Adding to Projects mandays." If the relative size is lower than that of the threshold, the newly discovered tasks are totally absorbed without triggering any adjustments to the project's man-days estimate. If the relative size exceeds the threshold value, part or all of the additional tasks are translated into addtional man-days in the projects plan. (p. 127-128)
aux FractOfMDForDevelopmentOnQA =
CumulativeQualityAssuranceMD/Cumulative_ManDayExpended
doc FractOfMDForDevelopmentOnQA = (man power allocation)
aux FullTimeEquivalentExperiencedWorkforce =
ExperiencedWorkforce*AverageDailyManpowerPerStaff
doc FullTimeEquivalentExperiencedWorkforce = Full-time equivalent experienced workforce
aux FullTimeEquivalentWorkforce = TotalWorkforceLevel*AverageDailyManpowerPerStaff
doc FullTimeEquivalentWorkforce = The amount of full time employees the workforce represents given daily manpower per staff.
(From human resource management subsystem)

aux	Identification_ManPoweNeeded_perTask =
TestingMapowerNeededPerTask*PercentIdentificationnManpower_PerTask
aux	IdentificationManPoweNeeded_PerError =
TestingManpowerNeeded_PerAverageError*PercentIdentificationManpower_PerError
doc	IdentificationManPoweNeeded_PerError =
TestingManpowerNeeded_PerAverageError*PercentLabelingManpower_PerError_1
aux	IndicatedCompletionDate = TIME +TimePercievedStillRequired
doc	IndicatedCompletionDate = "Time_percieved_still_required" is added to the value of "TIME", the number of working days elapsed on the project, to determine the "Indicated completion date". "Indicated completion date" is used to adjust the projects formal "Scheduled completion date", where "Indicated completion date" is the goal.
aux	IndicatedWorkforce = MDRemaining/(TimeRemaining + 0.001)/AverageDailyManpowerPerStaff
doc	IndicatedWorkforce = The "Indicated workforce level" represents the number of full time employees believed to be necessary and sufficient to to complete the project according to the current "Scheduled Completion Date". If employees are not workng full time, this is indicated by the "Average_daily_manpower_per_staff" which is less than one if people are working less than full time. (p.129)
aux	MaxDailyManPower_forClarification = IF(ReworkConsiderableTasks_Reported>0, (DailyManpowerForTesting),0)
doc	MaxDailyManPower_forClarification = IF(ReworkConsiderableTasks_Reported>0, (DailyManpowerForTesting*PercentManpower_for_Identification),0)		Armin! (IF(Manual=1)then(Classification_ManPower_Decided_by_Clients)else(DailyManpower_Allocated_ForTesting*Fraction_of_Effort_for_Classification))
aux	MaxDailyManPower_forClassification = IF(ReportedDeviation>0, (DailyManpowerForTesting),0)
doc	MaxDailyManPower_forClassification = Armin! (IF(Manual=1)then(Classification_ManPower_Decided_by_Clients)else(DailyManpower_Allocated_ForTesting*Fraction_of_Effort_for_Classification))
aux	MaxDailyManpower_forClientReview = IF(PresentedTasks_WaitingClientreview>0, DailyClientManpower_forClientReview,0)
aux	MaxDailyManPower_forIdentificationn = IF(ReworkConsiderableErrors_AfterSystemTesting>0, (DailyManpowerForTesting),0)
doc	MaxDailyManPower_forIdentificationn = IF(DetectedErrosinTesting_Density_1>0, (DailyManpowerForTesting*PercentEffort_for_Presentation),0)		Armin! (IF(Manual=1)then(Classification_ManPower_Decided_by_Clients)else(DailyManpower_Allocated_ForTesting*Fraction_of_Effort_for_Classification))

aux	MaxDailyManPower_forPresentation = IF(NoReworkConsiderableTasks_Reported>0, (DailyManpowerForTesting),0)
doc	MaxDailyManPower_forPresentation = IF(DetectedErrosinTesting_Density_1>0, (DailyManpowerForTesting*PercentEffort_for_Presentation),0)			Armin!
(IF(Manual=1)then(Classification_ManPower_Decided_by_Clients)else(DailyManpower_Allocated_ForTesting*Fraction_of_Effort_for_Classification))
aux	MaxDailyManPower_forTagging = IF(ReworkedErrors>0, (DailyManpowerAllocatedForQA),0)
doc	MaxDailyManPower_forTagging = IF(DetectedErrosinTesting_Density_1>0, (DailyManpowerForTesting*PercentEffort_for_Presentation),0)			Armin!
(IF(Manual=1)then(Classification_ManPower_Decided_by_Clients)else(DailyManpower_Allocated_ForTesting*Fraction_of_Effort_for_Classification))
aux	MaximumShortageIMDThatCanBeHandled = (OverworkDurationThreshold*FullTimeEquivalentWorkforce*MaxBoostInManHours)*WillingnessToWorkOvertime
doc	MaximumShortageIMDThatCanBeHandled = The Maximum shortage in MD that may be handled by a boost in manpower is the Workforce (Man) * Overwork Duration Threshold (days) * the Max Boost in Man Hours (Dimensionless) * Willingness to work overtime (1/0)

aux	MaximumTolerableCompletionDate = TotalDevelopmentTime*MaxScheduleCompletionDateExtension
aux	MDExcessesThatWillBeAbsorbed = MAX(0,(GRAPH(TotalMDPercievedStillNeeded/MDRemaining,0,0.1,[0,0.2,0.4,0.55,0.7,0.8,0.9,0.95,1,1,1"Min:0;Max:1"])*MDRemaining-TotalMDPercievedStillNeeded))
doc	MDExcessesThatWillBeAbsorbed = GRAPH* MDRemaining constitutes the MD Reported Still Needed!
(The GRAPH y-axis is MD Reported Still Needed / MD Remaining)
By subtracting the Total MD (Actually) Perceived Still Needed, we are left with MD Excesses that has been decided should be absorbed.
aux	MDHandledOrAbsorbed = IF(PercievedShortageInMD>=0,MIN(PercievedShortageInMD,MaximumShortageIMDThatCanBeHandled),-MDExcessesThatWillBeAbsorbed)*ControlSwitchForOverworkPolicy
doc	MDHandledOrAbsorbed = a. When the project is perceived to be behind schedule, the staff needs to handle the perceived shortage, - unless that shortage exceeds what maximally may be handled b. When the project is ahead of schedule, some MD Excesses will be absorbed, - giving rise to a negative number!

aux	MDPercievedNeededToReworkDetectedErrors = DetectedErrors_WaitingforRework*PercievedReworkManpowerNeededPerError

doc	MDPercievedNeededToReworkDetectedErrors = The number of mandays percieved still needed to rework allready detected errors is found by multiplying the number of detected errors with the the manpower needed per error.

aux	MDPercievedRemainingForNewTasks = MAX(0, MDRemaining - MDPercievedNeededToReworkDetectedErrors - MDPercievedStillNededForTesting)

doc	MDPercievedRemainingForNewTasks = Mandays percieved remaining for new tasks is the total number of mandays remaining minus mandays percieved needed for rework and testing.

aux	MDPercievedStillNededForTesting = TasksRemainingToBeTested/PercievedTestingProductivity

doc	MDPercievedStillNededForTesting = Given the percived testing productivity and the tasks remaining to be tested, this equation shows how the perception about mandays still needed for testing is found.

aux	MDPercievedStilNeededForNewTasks = NewTasksPercievedRemaining/AssumedDevelopmentInclQAProductivity

doc	MDPercievedStilNeededForNewTasks = As progress is measured during the rate at which resources are xpended, status reporting ends up being nothing more than an echo of the original plan. In other words "man days percived still needed for new tasks" will be equal to "man-days percieved remaining for new tasks". The project develops this chages as discrepancies between used and rmaining resources becomes moew apparent. Instead it becomes a function of what the project memebers percieve to be the amount of work that is remaning. (p.119)

aux	MDRemaining = MAX(0.0001, TotalJobSizeInMDs-Cumulative_ManDayExpended)

doc	MDRemaining = Mandays remaining to be used form the manday budget, that is the "total jobsize in mandays".	(From control subsystem)

aux	MDReportedStillNeeded = MDRemaining+ShortageInMDReported

doc	MDReportedStillNeeded = Each time this variable is different form "Man_days_remaining", that is, the variable "Reported_shortage_in_man_days_still_needed" is different from zero, this constitutes a revision of what the project's scope is percieved to be. (p.123-124)(control)

aux	MulltiplToErrGenDueToWorkforceMix = GRAPH(FractExperWorkforce,0,0.2,[2,1.8,1.6,1.4,1.2,1"Min:1;Max:2"])

doc	MulltiplToErrGenDueToWorkforceMix = This is a sum of several factors on error generatin rate. If the total workforce is less experienced, they will create more errors than if there was only experienced ones in the project (i.e. multiplier =1 which means normal error generation). (p100-101)

aux	MultiplDueToManpowerProductivity = GRAPH(DevelopmentProductivity,0,0.3,[0.01,0.3,0.6,0.9,1.2,1.5,1.8,2.1,2.4,2.7,3"Min:0;Max:3"]) aux	MultiplierToActiveErrorRegenDueToErrorDensity = GRAPH(AveActivErrorDensityPerDSI,0,10,[1,1.1,1.2,1.325,1.45,1.6,2,2.5,3.25,4.35,6"Min:0;Max:6"])

doc	MultiplierToActiveErrorRegenDueToErrorDensity = Multiplier to Active Error Regeneration due to Error Density. As the error density increases, the distribution of errors among the system's modules generally also increase; errors become less localized, and more expencive to detect and correct. This means increasing costs. (p 113)

aux	MultiplierToDetectionEffortDueToErrorDensity = GRAPH(ErrorDensityDSI,0,1,[50,36,26,17.5,10,4,1.75,1.2,1,1,1"Min:0;Max:50"])

doc	MultiplierToDetectionEffortDueToErrorDensity = Effect on the detection effort due to error density.

aux	MultiplierToPotentialProductivityDueToLearning = GRAPH(PercentOfJobActuallyWorked,0,0.1,[1,1.0125,1.0325,1.055,1.09,1.15,1.2,1.22,1.245,1.25,1.25"Min:0.9;Max:1.3"])

doc	MultiplierToPotentialProductivityDueToLearning = This is the learning curve effect. It is S-shaped and a function of progress in the project, starting with a value of 1 at the beginning of the project and peaking at a value 25% higher (i.e. 1.25) towards the end of the development period.

aux	MultiplierToProductivityDueToMotivationAndCommunicationLosses = ActualFractionOfAManDayOnProject*(1-CommunicationOverhead)

doc	MultiplierToProductivityDueToMotivationAndCommunicationLosses = This variable represents the average productive fraction of a man-day. Communication losses refer to project-type communication losses, and are thus formulated as a fraction of the "project hours" devoted to project work, hence the multiplicative formulation of the two components of productivity loss.(p.84)
(From software development productivity subsector)

aux	MultiplierToReworkEffortDueToDetectedErrorDensity = GRAPH(DetectedErrorDensityDSI,0,1,[1.3,1.153,1.099,1.068,1.045,1.035,1.019,1.014,1.007,1,1"Min:1;Max:1.5"])

doc	MultiplierToReworkEffortDueToDetectedErrorDensity = GRAPH(DetectedErrorDensityDSI,0,1,[3,2.29,1.86,1.56,1.35,1.23,1.13,1.06,1,1,1"Min:0;Max:5"])
Armin: More density of erros means more typographical and small erros. less denisty of erros means logical and big technical erros. --- S: When a subtle error is detected, its awareness is know. The effort to find out the nature of the error and how to rework it remains. Its not only "here is the error" but "Why is it here?" However, the reason that the multiplier for the QA effort is higher than the rework effort rests on the nature of the effort.   When error density is low, the QA effort is very hard. QA are trying to find something that is subtle in a vast emtpty room. High density reveals error-patterns that can give the design-error away, when no such pattern excist its like finding the needle in the haystack.

Rework on the other hand already know where the error is as QA detected it, and the effort is subsequently lower in the initial effort at low density.

aux	MultiplierToTestingEffortDueToErrorDensity = GRAPH(TotalErrorDensityDSI,0,0.1,[1.1,1.049,1.024,1.014,1.011,1.008,1.006,1.003,1.001,1.001,1.001"Min:1;Max:1.2"])

doc	MultiplierToTestingEffortDueToErrorDensity = GRAPH(TotalErrorDensityDSI,0,2,[1.2,1.12,1.07,1.03,1.01,1"Min:1;Max:1.2"]) GRAPH(TotalErrorDensityDSI,0,2,[1.5,1.29,1.14,1.05,1,1"Min:0;Max:2"])

aux	MultiplToBadFixesGenDueTo_WorkforceMix = GRAPH(FractExperWorkforce,0,0.2,[1.3,1.228,1.164,1.099,1.048,1"Min:1;Max:1.5"])

doc	MultiplToBadFixesGenDueTo_WorkforceMix = GRAPH(FractExperWorkforce,0,0.2,[1.3,1.154,1.064,1.031,1.011,1"Min:1;Max:1.5"])

aux	MultiplToBdFixesGenDueTo_ErrorType = GRAPH(PercentOfJobActuallyWorked,0,0.1,[0.15,0.149,0.143,0.133,0.113,0.075,0.04,0.019,0.008,0.004,0.002"Min:0;Max:0.15"])

doc	MultiplToBdFixesGenDueTo_ErrorType = GRAPH(PercentOfJobActuallyWorked,0,0.1,[0.075,0.0743,0.0729,0.069,0.0612,0.0519,0.0453,0.0415,0.0398,0.0385,0.0379"Min:0.03;Max:0.075"])

aux	MultiplToBdFixesGenDueTo_SchedulePressure = GRAPH(SchedulePressure,-0.4,0.2,[0.9,0.947,1,1.032,1.071,1.105,1.145,1.2"Min:0.9;Max:1.5"])

doc	MultiplToBdFixesGenDueTo_SchedulePressure = GRAPH(SchedulePressure,-0.4,0.2,[0.9,0.946,1,1.03,1.067,1.1,1.144,1.202"Min:0.9;Max:1.3"])

aux	MultiplToErrGenerationDueToSchedulePressure = GRAPH(SchedulePressure,-0.4,0.2,[0.9,0.94,1,1.05,1.14,1.24,1.36,1.5"Min:0.9;Max:1.5"])

doc	MultiplToErrGenerationDueToSchedulePressure = People under time pressure don't work better, they just work faster. People make more error with more stress, activities start overlapping when they should have been sequential. Under no schedule pressure the multiplier is 1, but errror generation can increase with as much as 50% (i.e. 1,5), when ahead of schedule error generation is less than normal (p.101)

aux	Mutiplierto_NoReworkConsiderableErrors_DuetoSchedulePressure = GRAPH(SchedulePressure,-0.4,0.2,[0.9,0.94,1,1.05,1.14,1.24,1.36,1.5"Min:0.9;Max:1.5"])

aux	MutipltoBadFixGenDueto_DetectedErrorDensity = GRAPH(DetectedErrorDensityDSI,0,2,[1,1.02,1.053,1.096,1.169,1.3"Min:1;Max:1.5"])

doc	MutipltoBadFixGenDueto_DetectedErrorDensity = GRAPH(DetectedErrorDensityDSI,0,2,[1,1.03,1.07,1.11,1.19,1.3"Min:1;Max:1.5"])	Armin: Due to tiredness and lso ess (motivation!!!)

aux MutipltoBadFixGenDueto_ReworkedErrorDensity = GRAPH(ErrorsReworkedDuringDevelopmen_perTask,0,1,[1,0.759,0.604,0.518,0.449,0.406,0.369,0.343,0.326,0.313,0.303"Min:0;Max:1"])
doc MutipltoBadFixGenDueto_ReworkedErrorDensity = GRAPH(ErrorsReworkedDuringDevelopmen_perTask,0,1,[1,0.64,0.41,0.27,0.19,0.13,0.09,0.06,0.04,0.02,0"Min:0;Max:1"])
aux NewTasksPercievedRemaining = CurrentlyPerceivedJobSize-CumulativeTasksDeveloped
doc NewTasksPercievedRemaining = This is the amount of tasks percieved still remaining, given that all tasks may not have been discovered yet.
(From control subsystem)
aux NominalNoOfErrorsCommitedPerKDSI = GRAPH(PercentOfJobActuallyWorked,0,0.2,[25,23.86,21.59,15.9,13.6,12.5"Min:10;Max:25"])
doc NominalNoOfErrorsCommitedPerKDSI = This is measured in KDSI instead of tasks, since most publised data on error rates are in terms of KDSI. We will asume in the model that the development phase will be didvided equally between design and coding actvities. (p.98) The number of errors will be smaller/KDSI as the project progresses. It wil go from 25 errors/KDSI in the beginning to 12.5 at the end. The half way (0,5) is the division point between design and coding.
aux NominalNoOfErrorsCommittedPerTask = NominalNoOfErrorsCommitedPerKDSI*DSIPerTask/1000
doc NominalNoOfErrorsCommittedPerTask = The cumulative effect of both organisational factors and the quality of the staff is represented in the model as a single nominal value, "Nominal errors committed per task. This is a variable as type and number of errors differ during a projects different stages. Average rate for design = 23 and coding = 14.5 (p98-99)
aux NominalOverworkDurationThreshold = GRAPH(TimeRemaining,0,10,[0,10,20,30,40,50"Min:0;Max:50"])
doc NominalOverworkDurationThreshold = The nominal value for the "overwork duration threshold" is set at 50 working days (i.e. 10 weeks). (p.87)
aux NominalReworkManpowerNeededPerError = GRAPH(PercentOfJobActuallyWorked,0,0.2,[0.6,0.575,0.5,0.4,0.325,0.3"Min:0.3;Max:0.6"])
doc NominalReworkManpowerNeededPerError = This nominal component is a function of error-type: design or coding (i.e. % of job actually worked) Design errors, are generated at a higher rate and are more costly to detect, and aslo more costly to rework. (p.107)
aux NominalTestingManpowerNeededPerError = GRAPH(PercentOfJobActuallyWorked,0,0.2,[0.3,0.296,0.28,0.218,0.204,0.2"Min:0.2;Max:0.3"])
doc NominalTestingManpowerNeededPerError = GRAPH(PercentOfJobActuallyWorked,0,0.2,[0.15,0.1487,0.1454,0.1351,0.1311,0.13"Min:0.13;Max:0.15"])

GRAPH(PercentOfJobActuallyWorked,0,0.2,[0.1197,0.118,0.1132,0.1048,0.1013,0.1004"Min:0.05;Max:0.15"])

aux	NomQAManpowerNeededToDetectAverageError =
GRAPH(PercentOfJobActuallyWorked,0,0.1,[0.4,0.4,0.39,0.375,0.35,0.3,0.25,0.225,0.21,0.2,0.2"Min:0.15;Max:0.45"])

doc	NomQAManpowerNeededToDetectAverageError = Design errors are not only generated at a higher rate, but they are also more costly to detect than coding errors. This can be seen from the graph as the project progresses (p.103)

aux	NormalWorkRateAdjustmentDelay =
GRAPH(TimeRemaining,0,5,[2,3.5,5,6.5,8,9.5,10"Min:0;Max:30"])

doc	NormalWorkRateAdjustmentDelay = Normal Work Rate Adjustment Delay

aux	OverworkDurationThreshold =
EffectOfExhaustionOnOverWorkDurationThreshold*NominalOverworkDurationThreshold

doc	OverworkDurationThreshold = Overwork Duration Threshold.
The multiplier adjust this threshold downwards. This formulation differentiate between how hard employees work. The multiplier induce more cut in the threshold with harder periods of work. Once people start working harder, this variable represents the maximum remaining duration for which they are willing to continue working harder decreases below the nominal value.(how many days they are willing to work overtime)

aux	PassiveErrorDensityPerTask = UndetectedPassiveErrors/(TasksQAed + 0.0001)

doc	PassiveErrorDensityPerTask = The number of undetected passive errors per tasks QAed.

aux	PerceivedDesignAndCodingProductivity =
CumulativeTasksDeveloped/CumulativeCodingDesignMD

aux	PerceivedProjectCostsInMD =
Cumulative_ManDayExpended+TotalMDPercievedStillNeeded

aux	PerceivedSizeOfDiscoveredTasksMD = (IF(TasksDiscovered<0, 0, TasksDiscovered))/AssumedDevelopmentInclQAProductivity

doc	PerceivedSizeOfDiscoveredTasksMD = The percieved size of discovered tasks in mandays, is defined as the number of discovered tasks divided by the assumed productivity. The IF-sentence is to avoid negative values for tasks discovered which may occur due to integration-error in Powersim.

aux	PercentAdjustmentInPlannedFractionOfManpowerForQA =
GRAPH(SchedulePressure,0,0.1,[0,-0.025,-0.15,-0.35,-0.475,-0.5"Min:-0.6;Max:0.1"])

doc	PercentAdjustmentInPlannedFractionOfManpowerForQA = Adjustment in planned fraction of manpower for quality assurance. Increased schedulpressure reduce fraction of manpower for quality assurance. i.e effort percieved remaining > project budget -> Schedulpressure increases. The relation between schedulepressure and quality assurance effort is an estimation. Planned QA can be relaxed as much as 50% with increased schedulepressure aux PercentBoostInWorkRateSought = IF(PercievedShortageInMD >= 0, (MDHandledOrAbsorbed/(FullTimeEquivalentWorkforce*(OverworkDurationThreshold+0.0001))), (MDHandledOrAbsorbed/(TotalMDPercievedStillNeeded-MDHandledOrAbsorbed+0.0001)))

doc PercentBoostInWorkRateSought = Percentage Boost in Work Rate Sought to;

a. handle Perceived Shortages in MD; or

b. absorb Perceived Excesses in MD:

a. When project is perceived to be behind schedule, i.e. total effort still needed to to complete project is perceived to be greater than total effort actually remaining, then there is a Percentage Boost in (addition to) Work Rate Sought:

WorkrateSought = Nominal Fraction of AMDOnProject *(1+PercentBoost ...).

It is really not perceint, but a fraction of the potential overwork:

The value of the man-days to be handled, divided by the product of

the fulltime equivalent workforce and overwork duration threshold (how many days they are willing to work overtime). For Example, if 100 MD is to be handled by 10 Men in the course of 50 Days, then this implies a fractional increase of their effort in the amount of 100MD / (10 M * 50 D) = 0.2 MD / MD. So by increasing their work rate by 20% (*1.2), the backlogged work can be handled over 50 days.

b. The Denominator (TotalMDPercievedStillNeeded-MDHandledOrAbsorbed) corresponds to the MDReported Still Needed. So the negative boost is the negative fraction of the MD Reported Still Needed that will be absorbed through a reduction in Workrate Sought.

aux PercentErrorsDetected = 100*(CumulativeDetectedErrorsWaitingforRework+Cumulative_DetectedErrorsinTesting)/(CumulativeErrorsGeneratedDirectlyDuringWork+0.001)

doc PercentErrorsDetected = The number of errors detected relative to the number of errors generated directly during working.

aux PercentErrorsReworked = 100*CumulativeDetectedErrors_Reworked/(CumulativeDetectedErrorsWaitingforRework+Cumulative_DetectedErrorsinTesting+0.001)

aux PercentOfJobActuallyWorked = CumulativeTasksDeveloped/RealJobSizeTasks

doc PercentOfJobActuallyWorked = The percent of job actually worked is defined as the cumulative tasks developed divided by the real jobsize in task. (From control subsystem)

aux PercentOfJobPerceivedWorked = (CumulativeTasksDeveloped/CurrentlyPerceivedJobSize)*100

doc PercentOfJobPerceivedWorked = Multiplying by hundred turns the fraction into percentage.(control subsystem)

aux PercentOfTasksPerceivedTested = CumulativeTasksTested/CurrentlyPerceivedJobSize

doc PercentOfTasksPerceivedTested = Percent of Tasks Tested

aux PercentOfUndiscoveredTasksDiscoveredPerDay =
GRAPH(PercentOfJobPerceivedWorked,0,20,[0,0.4,2.5,5,10,100"Min:0;Max:100"])
doc PercentOfUndiscoveredTasksDiscoveredPerDay = Because the teams level of knowledge of what the software product is intended to do increases with project development, (p.126), so does also the percentage which is dependent on "Percent_of_Job_Percived_Worked". The slope of the table function rises slowly until 80% of Percent_of_Job_Percived_Worked, and then increases sharply until it reaches 100%.
aux PercievedDevelopmentInclQAProductivity =
CumulativeTasksDeveloped/(Cumulative_ManDayExpended-CumulativeTestingMD)
doc PercievedDevelopmentInclQAProductivity = Percieved development productivity is defined as the number of task developed so far per number of mandays used so far.
aux PercievedJobSizeI_DSI = RealJobSizeDSI*(1-TaskUnderestimationFraction)
doc PercievedJobSizeI_DSI = (From initialistaion sector)
aux PercievedShortageInMD = TotalMDPercievedStillNeeded-MDRemaining
doc PercievedShortageInMD = Percieved shortage or excess in mandays shows whether the project is percieved to behind or ahead of schedule.
(From control subsystem)
aux PlannedFractionOfManpowerForQA =
(GRAPH(PercentOfJobActuallyWorked,0,0.1,[0.15,0.15,0.15,0.15,0.15,0.15,0.15,0.15,0.15,0.15,0"Min:0;Max:1"])) * (1+QualityObjective/100)
doc PlannedFractionOfManpowerForQA =
(GRAPH(PercentOfJobActuallyWorked,0,0.1,[0.15,0.15,0.15,0.15,0.15,0.15,0.15,0.15,0.15,0.15,0"Min:0;Max:1"])) * (1+QualityObjective/100) Planned fraction of manpower for quality assurance. Planned fraction of manpower for QA is 15% of total daily manpower as long as all the work is not finished. This is indicated by the last number in the table-function as it is zero. It is also possible to use an IF-function here.
aux PlannedTestingProductivity =
CurrentlyPerceivedJobSize/PlannedTestingSizeInMDBeforeWeStartTesting
aux PotentialClassification_rate =
ActualDailyManPower_forClassification/ClassificationManPoweNeeded_perTask
aux PotentialClientManpowerNeededtoReview_PerTask =
ClientManpowerNeeded_PerTask*PercentClienReviewingManpower_PerTask
aux PotentialClientReviewing_rate =
ActualDailyManPower_forClientReview/PotentialClientManpowerNeededtoReview_PerTask
aux PotentialDevelopmentProductivity =
AverageNominalPotentialProduvtivity*MultiplierToPotentialProductivityDueToLearning

doc	PotentialDevelopmentProductivity = Potential productivity is the level of productivity that will be attained if the individual or group makes the best possible use of the aviliable resources.
aux	PotentialErrorDetectionRate =
DailyManpowerAllocatedForQA/QAManpowerNeededToDetectAverageError
doc	PotentialErrorDetectionRate = Potentially Error detection rate represents the maximum number of errors that can be detected at one time and is determined by dividing the value of the QA effort allocated by the value of the QA effort that is needed on average to detect an error. (p.103)
aux	PotentialIClarification_Rate =
ActualDailyManPower_forClarification/ClarificationManPowerNeeded_perTask
aux	PotentialIdentification_Rate =
ActualDailyManPower_forIdentificationn/Identification_ManPoweNeeded_perTask
aux	PotentialPresentation_Rate =
ActualDailyManPower_forPresentation/PresentationManPoweNeeded_PerTask
aux	PotentialReworkRate =
DailyManpowerAllocatedForRework/ReworkManpowerNeededPerAverageError
aux	PotentialTagReleasing_Rate =
ActualDailyManPower_forTagging/TaggingManPoweNeeded_PerError
aux	PresentationManPoweNeeded_PerTask =
TestingMapowerNeededPerTask*PercentPresentationManpower_PerTask
aux	ProjectedDesignAndCodingProductivity = ProjectedDevelopmentInclQAProductivity/(1-FractOfMDForDevelopmentOnQA)
aux	ProjectedDevelopmentInclQAProductivity =
NewTasksPercievedRemaining/(MDPercievedRemainingForNewTasks+0.1)
doc	ProjectedDevelopmentInclQAProductivity = Projected development productivity is defined as the task percived remaining divided by man-days percived remaining for new tasks. Thus the tasks are supposed to be finished by using the mandays percived remaining for these new tasks.(control)
aux	ProjectSize = CumulativeTasksDeveloped*DSIPerTask
aux	QAManpowerNeededToDetectAverageError =
NomQAManpowerNeededToDetectAverageError*(1/MultiplDueToManpowerProductivity)*MultiplierToDetectionEffortDueToErrorDensity
doc	QAManpowerNeededToDetectAverageError = The actual QA manpower needed to detect an error is a functin of errror-type, and how efficiently people work.(p. 103) Manpower is lost to communication and other activities. The more errors there are (higher error density) the easier it is to find and correct them. (p.105) The fewer errors, the harder it is to find them, and some will eventually escape to the subsequent phases of software development (p.106)
aux	RealJobSizeTasks = RealJobSizeDSI/DSIPerTask

aux RelativeSizeOfDiscoveredTasks =
PerceivedSizeOfDiscoveredTasksMD/(MDPercievedRemainingForNewTasks+0.0001)
doc RelativeSizeOfDiscoveredTasks = This is the relative size of the discovered tasks. (p.127).
aux ReportedDeviations_perTask =
MAX(CumulativeNoReworkConsiderableDeviation/(CumulativeTasksTested+0.0001), 0)
aux ReportedErrorsperTask = MAX(ReportedDeviation/(TasksQAed+0.0001), 0)
aux ReworkManpowerNeededPerAverageError =
NominalReworkManpowerNeededPerError*MultiplierToReworkEffortDueToDetectedErrorDensity*(
1/MultiplDueToManpowerProductivity)
doc ReworkManpowerNeededPerAverageError = The actual rework manpower that would be needed to correct an error, in addition to being a function of error-type, must also depend on how efficiently people work. That is we need to account for productivity losses due to communication and motivation. (From quality assurance and rework sector)
aux ScheduleAdjustmentTime = GRAPH(TimeRemaining,0,5,[0.5,5"Min:0;Max:5"])
doc ScheduleAdjustmentTime = "Schedule adjustment time" is set to 5 working days.
aux SchedulePressure = (TotalMDPercievedStillNeeded-MDRemaining)/MDRemaining
doc SchedulePressure = The hgigher the number the bigger is the shcedulepressure. If there are many mandays percieved still needed and few remaining this is the same as a higher schedulepressure. (From Control subsystem)
aux ShortageInMDReported = PercievedShortageInMD - MDHandledOrAbsorbed
doc ShortageInMDReported = "Reported shortage or excess in mandays" is determined by "Percieved_shortage_or_excess_in_man_days" which is subtracted by "Handled_or_absorbed_man_days" in the software development sector.
aux T = TIME
aux TaggingManPoweNeeded_PerError =
QAManpowerNeededToDetectAverageError*PercentTaggingManpower_PerError
aux TasksRemainingToBeTested = CurrentlyPerceivedJobSize-(CumulativeTasksTested)
doc TasksRemainingToBeTested = CurrentlyPerceivedJobSize-
(CumulativeTasksTested+CumulativeTasksClientTested) Tasks remaining to be tested is the currently percieved job size minus the total number of tasks tested so far.
aux TeamSize = (TotalMD/TotalDevelopmentTime)/AverageDailyManpowerPerStaff
aux TeamSizeAtTheBeginningOfDesign = TeamSize*InitialUnderstaffingFactor
doc TeamSizeAtTheBeginningOfDesign = The team size at the beginning of design is initially understaffed by 50% according to the litterature. p.157 (Part of the initialisation sector.)

aux TestingManpowerNeeded_PerAverageError = (NominalTestingManpowerNeededPerError*MultiplierToTestingEffortDueToErrorDensity)*(1/MultiplDueToManpowerProductivity)

doc TestingManpowerNeeded_PerAverageError = (NominalTestingManpowerNeededPerError*MultiplierToTestingEffortDueToErrorDensity)*(1/MultiplDueToManpowerProductivity) The amount of testing manpower needed per error. NominalTestingManpowerNeededPerError*

aux TestingMapowerNeededPerTask = ((TestingEffortOverhead*DSIPerTask/1000) *(1/MultiplDueToManpowerProductivity))+ (TestingManpowerNeeded_PerAverageError*TotalErrorDensity)

doc TestingMapowerNeededPerTask = "Normal tesing effort needed per task" has a fixed and a variable component. The variable component is a function of the number of errors in a task, and it represents the tesing effort that would be expended in the actual detection and correction of errors. The fixed component is independent on the number of errors. In addition comes the efficiency of testing, depending on motivation and communication losses.(p115-116)

aux TestingMD = (1-FractOfEffortAssumedNeededForDevelopment)*TotalMD

doc TestingMD = 20% of the total mandays is consumed in testing of the software.

aux TestingProductivity = IF(0 >= (CumulativeTasksTested), PlannedTestingProductivity, ActualTestingProductivity)

doc TestingProductivity = IF(0 >= (CumulativeTasksTested+CumulativeTasksClientTested), PlannedTestingProductivity, ActualTestingProductivity) As systen testing gets underway, peoples perceptions of their productivity become a function of how productive the testing activity actually is as opposed to how productive it was planned to be. (p.123)

aux TimePercievedStillRequired = MDRemaining/(WorkforceForceSought*AverageDailyManpowerPerStaff)

doc TimePercievedStillRequired = Dividing the "Man_days_remaining" by the "Workforce_force_sought" (in terms of full-time emplyees" determines the "Time percieved still required", which represents the remaining time, in working days, percieved to be required to complete the project, given its current condition. (p.134) Using "Workforce_force_sought" as an input, assumes that schedule adjustments are made with full awareness of the hiring decisions being made in the project. (p.134).

aux TimeRemaining = MAX(ScheduledCompletionDate - TIME, 0)

doc TimeRemaining = Subtracting the value of "TIME" (which represents the number of working days elapsed in a simulation run) from the "Scheduled_completion_date", we can determine the "Time remaining". (From planning subsector)

aux TotalDevelopmentMD = CumulativeQualityAssuranceMD+CumulativeReworkMD+CodingDesignTrainingMD

aux TotalDevelopmentTime =
ScheduleSwitch*((19*2.5*EXP(0.38*LN(TotalMD/19)))*ScheduleCompressionFactor)+(1-ScheduleSwitch)*TimeToDevelop
doc TotalDevelopmentTime = Schedule swith and time to develop makes it possible to change the initial schedule . Thus using another estimation method than COCOMO, for example based on experience from another run. (From initialisation sector)
aux TotalErrorDensity = ActiveErrorDensityPerTask+PassiveErrorDensityPerTask
doc TotalErrorDensity = The total error density is the sum of active and passive error density.
aux TotalErrorDensityDSI = TotalErrorDensity*1000/DSIPerTask
aux TotalMD = MDSwitch*(((2.4*EXP(1.05*LN(PercievedJobSizeI_DSI/1000)))*19)*(1-MDUnderestimationFraction)) + (1-MDSwitch)*TrueTotalMD
aux TotalMDPercievedStillNeeded =
MDPercievedStilNeededForNewTasks+MDPercievedStillNededForTesting+MDPercievedNeededToReworkDetectedErrors
doc TotalMDPercievedStillNeeded = "Total mandays percieved still needed" to complete the project are determined from the measurements. This variable includes mandays to develop QA tasks, to rework and complete system testing.(p. 117) At any point the work that will be percieved remaining will in general be a combiantion of three things:(1) Work to develop and QA new tasks, (2) rework of any detected errors, (3) system testing work (p.117) (From control subsystem)
aux TotalWorkforceLevel = ExperiencedWorkforce+NewWorkforce
doc TotalWorkforceLevel = Total number of workers. Is a part of two important factors: 1. planning of the needed workforce 2. stability of the workforce.
(From the human resource management subsystem.)
aux TransferRate = MAX(0,-WorkforceGap/DelayToTransferPeopleOut)
doc TransferRate = If there is a need to transfer people out of the project this is indicated by the negative number of the workforce gap divided by the time it takes to transfer people out.
aux Undetected_Errors = UndetectedActiveErrors+UndetectedPassiveErrors
aux WCWF_1 =
GRAPH(TimeRemaining/(HiringDelay+AverageAssimilationDelay),0,0.3,[0,0,0.1,0.4,0.85,1,1,1,1,1,1"Min:0;Max:1"])
doc WCWF_1 = "Willingness to change workforce level 1" is one of two components and represents the pressures that develop for work force stability as the project proceeds toward its final stages. (p.131) . As the number of days percieved remaining drops below 1.5 * (Hiring_delay+Average_assimilation_delay), there is a reluctance to increase the workforce level. (p.132)

aux WCWF_2 =
GRAPH(ScheduledCompletionDate/MaximumTolerableCompletionDate,0.86,0.02,[0,0.1,0.2,0.35,0.6,
0.7,0.77,0.8"Min:0;Max:1"])
doc WCWF_2 = As long as "Scheduled_completion_date" is comfortably below the "Maximum_tolerable_completion_date", the value of "Willingness to change workforce 2" is zero; that is, it has no bearing on the determination of the WCWF and consequently no bearing on the hiring decisions. When "Scheduled_completion_date" starts approaching the "Maximum_tolerable_completion_date" the value of WCWF2 starts to rise gradually. This happens at the end of a project, and as the WCWF2 exceeds WCWF 1, the "willingness to change workforce level" becomes totally dominated by scheduling with a goal not to overshoot the "Maximum Tolerable Completion Date." (p. 133)
aux WeightOnProjectedProductivity =
EffOfResourceExpenditureProjectedProdWeight*EffectOfProgressOnWeightOnProjectedProductivity
doc WeightOnProjectedProductivity = Only when the program is almost finished or when the allocation time budget is almost used up will the programmer be able to recognize the discrepancy between the percent of tasks accomplished and the percent of resources expended. (p.121) The weighting factor is the product of the two multipliers; due to development and resource expenditure. They move from one at the beginning of the project to zero at the end when one of them is zero.(p.121)
aux WillingnessToChangeWorkforceLevel = MAX(WCWF_1,WCWF_2)
doc WillingnessToChangeWorkforceLevel = Considerations is also given to the stability of the workforce. That is, before hiring new project members, management tries to estimate how long new members will be needed. In general, the realtive wheighting of the desire to have a stable work force and the desire to complete the project on time changes from stage to stage of the project. (p129-130) It is a variable that assumes values between 0 and 1. When "WCWF"=1, the weighting only considers the "Indicated work force level"; management is adjusting its workforce level to the number percived required to finish on schedule. As WCWF moves towards 0, more and more weigthing is given to the stability of the work force. When WCWF equals 0, the weighted number of employees desired is wholly dependent on work force stability. (p.131)
aux WillingnessToWorkOvertime = IF(TIME >=
TimeOfLastExhaustionBreakdown+TimeToRecover,1,0)
doc WillingnessToWorkOvertime = Willingness to Overwork (0 or 1)
During the "de-exhausting" period the work force remains unwilling to "reoverwork". This variable is sett to 0 when maximum exhaustion level is reached and the overwork duration threshold is driven to zero. (p.89). As long as the "Maximum Shortage in Man-Days to be Handled" is also zero.
aux WorkforceForceSought = MIN(CeilingOnTotalWorkforce,WorkforceLevelNeeded)

doc	WorkforceForceSought = The workforce level sought is the number of people that is needed (workforce needed), but this number is restricted by the ceiling on total workforce. Work force level sought is almost always equal to "Workforce_level_needed", but they can differ (usually at the beginning of a project), when the project's manpower buildup rate tends to be at its highest level. The "workforce level sought" in effect defines a ceiling on the number of employees to be hired.
(From human resource management subsystem)
aux	WorkforceGap = WorkforceForceSought-TotalWorkforceLevel
doc	WorkforceGap = The gap between workforce level sought (the desired goal) and the total workforce level (actual). If this number is positive, more people are needed. If it is negative less people are neded.
aux	WorkforceLevelNeeded =
MIN((WillingnessToChangeWorkforceLevel*IndicatedWorkforce+(1-WillingnessToChangeWorkforceLevel)*TotalWorkforceLevel),IndicatedWorkforce)
doc	WorkforceLevelNeeded = The "Work force level needed" is a weighted average of the current "Total Work Force Level" and the "Indicated_workforce_level". It accounts for the stable work force level and the number of employees that would be required to complete the project on time. This formulation applies only when the value of the "Indicated_workforce_level" is larger than "Total_workforce", indicating a need for hiring more people. When the opposite is true, "Workforce level needed" would simply be set to the lower value, and any excessive employees transferred out of the project. (p.131) (Part of the Planning sector)
aux	WorkRateAdjustmentDelay =
EffectOfWorkrateSoughOn_WRAD*NormalWorkRateAdjustmentDelay
doc	WorkRateAdjustmentDelay = This is a variable delay for the adjustment of the "Actual fraction of a man day on project."
aux	WorkrateSought = (1+PercentBoostInWorkRateSought)*NominalFractionOfAMDOnProject
const	AveDelayInIncorporatingDiscoveredTasks = 10
doc	AveDelayInIncorporatingDiscoveredTasks = It takes on average 10 days to incorporate discovered tasks into project.
const	Average_employment_time = 673
doc	Average_employment_time = The average time an employee is employed.
const	AverageAssimilationDelay = 80
doc	AverageAssimilationDelay = The time it takes to assimilate newly hired work force.
(From human resource management subsystem)
const	AverageDailyManpowerPerStaff = 1
doc	AverageDailyManpowerPerStaff = The average daily manpower per staff.
(From manpower allocation sector)
const	AverageQATime = 10/3

doc AverageQATime = Software tasks will always be QAed or considered QAed after a delay that is assumed to be independent of the QA effort. The average QA delay period is set at 2 weeks (i.e. 10 working days) (p.103)

const ClientManpower_Productivity = 1

doc ClientManpower_Productivity = .50

const ControlSwitchForOverworkPolicy = 1

doc ControlSwitchForOverworkPolicy = Allows us to test policy of no overwork.

const DailyClientManpower_forClientReview = 2

doc DailyClientManpower_forClientReview = 2

const DelayToTransferPeopleOut = 10

doc DelayToTransferPeopleOut = Average time delay to transfer people out of the project.

const DesiredClarification_Delay = 6

doc DesiredClarification_Delay = 1

const DesiredClassification_Time = 3

doc DesiredClassification_Time = 1

const DesiredClientReview_Delay = 3

doc DesiredClientReview_Delay = 1

const DesiredIdentificationn_Time = 3

doc DesiredIdentificationn_Time = 1

const DesiredPresentation_Time = 3

doc DesiredPresentation_Time = 1

const DesiredReworkDelay = 15

doc DesiredReworkDelay = The desired delay it takes to do rework.

const DesiredTagging_Time = 3

doc DesiredTagging_Time = 1

const DeviationReportPrepration_Time = 3

doc DeviationReportPrepration_Time = 2

const DSIPerTask = 60

doc DSIPerTask = The definition of one task is 60 DSI.

(From initialisation sector)

const ExhaustionDepletionTime = 20

doc ExhaustionDepletionTime = The average time it takes to recover form exhaustion

const FractOfEffortAssumedNeededForDevelopment = 0.80

doc FractOfEffortAssumedNeededForDevelopment = The percent of the distribution between the different phases. This means 80% to development and 20% to testing.

const HiringDelay = 40

doc	HiringDelay = The average time it takes to hire new workforce.

(From human resource management subsystem)

const	Init_SCD = INIT(ScheduledCompletionDate)

const	InitialUnderstaffingFactor = 0.54

const	MaxBoostInManHours = 1

doc	MaxBoostInManHours = The maximum boost that people can increase their working rate. In this case it can be boosted by max 100%=1.

const	MaximumTolerableExhaustion = 50

doc	MaximumTolerableExhaustion = The maximum tolerable level of exhaustion. At this point people will refuse to work overtime.

const	MaxNewHireesPerFulltimeExperiencedStaff = 3

doc	MaxNewHireesPerFulltimeExperiencedStaff = The largest number of new hirees that a single full-time experienced staff can be expected to handle effectivly.

const	MaxRelSizeOfAdditionsToleratedWithoutAddingToProjectMD = 0.01

doc	MaxRelSizeOfAdditionsToleratedWithoutAddingToProjectMD = "Maximum relative Size of Additions Tolerated Without Adding to Projects manday" is set to 1%. For example, for a 1000 man-day development phase (10 people working for 100 working days) the threshold is 10 man-days.

const	MaxScheduleCompletionDateExtension = 10000000

doc	MaxScheduleCompletionDateExtension = By giving this constant a high value, it is possible to represent projects in which there are no tight time commitments. (p.133) In that case WCWF is a function of WCWF1.

const	MDSwitch = 1

const	MDUnderestimationFraction = 0

const	NoConsiderableDeviation_ReportPrepration_Time = 3

doc	NoConsiderableDeviation_ReportPrepration_Time = 4, 2

const	NominalFractionOfAMDOnProject = 0.6

doc	NominalFractionOfAMDOnProject = Nominal Fraction of a Man-day on Project. The amount of time devoted on project work. I.e. a full-time employee is 8 hours, and daily contribution to the project will be less than 8 hours (normal 60%).

const	NominalPotentialProductivityOfExpEmployee = 1

doc	NominalPotentialProductivityOfExpEmployee = A certain number of tasks/ man-day. The nominal (actual) productivity for the average experienced staff-member.

const	NominalPotentialProductivityOfNewEmployee = 0.5

doc	NominalPotentialProductivityOfNewEmployee = A certain number of tasks/ man-day. The nominal (actual) productivity for the average newly employee.

const	One = 1

const	PercentBadFixes = 0.075

doc PercentBadFixes = Percent Bad Fixes

const PercentClassificationManpower_PerError = .10

const PercentClassificationManpower_PerTask = .10

const PercentClienReviewingManpower_PerTask = .1

doc PercentClienReviewingManpower_PerTask = 0.1

const PercentIdentificationManpower_PerError = .10

const PercentIdentificationnManpower_PerTask = .10

const PercentPresentationManpower_PerTask = .1

doc PercentPresentationManpower_PerTask = 0.1

const PercentTaggingManpower_PerError = .10

const QualityObjective = 0

doc QualityObjective = Normal Quality objective is = 0. The garantee that the product meets the specified standards.

const RealJobSizeDSI = 64000

doc RealJobSizeDSI = The total amount of work in the project.

const ReportingDelay = 10

doc ReportingDelay = Delay for smoothing the information about "percent of tasks reported complete" and "percent of development percieved complete".

const ReworkConsiderableTask_ReportingTime = 2

const ScheduleCompressionFactor = 1

const ScheduleSwitch = 1

const TaskUnderestimationFraction = 0.33

doc TaskUnderestimationFraction = This is a fraction that determines how much the estimated jobsize in tasks should be underestimated at initialisation. p. 156

const TestingEffortOverhead = 1

doc TestingEffortOverhead = If we assume that motivation and communication losses will result in a 50% loss in productivity, we have a documented overhead of 2 man-days/KDSI transformed into 1man-day/KDSI.

const TestingManpowerNeededperError = 0.15

doc TestingManpowerNeededperError = The amount of testing manpower needed per error.

const TimeToAdjust_P_RW_MP_N_P_Error = 10

doc TimeToAdjust_P_RW_MP_N_P_Error = The average time it takes to change ones perception about the rework manpower needed per error.

const TimeToAdjustPerceivedTestProd = 50

doc TimeToAdjustPerceivedTestProd = Time to persist a change in the actual testing productivity.

const TimeToDevelop = 0

const TimeToSmoothActiveErrorDensity = 40

doc	TimeToSmoothActiveErrorDensity = Time to Smooth Active Error Density. The average delay it takes to generate new errors from errors already commited is 3 months or 40 working days.
const	TrainersPerNewEmployee = 0.20
doc	TrainersPerNewEmployee = On the average each new emplyoee consumes in training overhead the equivalent of 20% of an experienced employee's time for the duration of the training or assimilation period.
const	TrueTotalMD = 0
const	Zero = 0

Appendix 2: Summary of the book "Software Project Dynamics- An Integrated Approach"

Appendix 4 contains my summary of the book *"Software Project Dynamics- An Integrated Approach"* published by Tarek Abdel-Hamid following his doctoral thesis in 1991. This book provides the foundation for the integrated system dynamics model utilized in my thesis.

Summary Chapter 1

Introduction

Since the late 70's and early 80 the demand and investments in software development has increased incrementally with the progress and development of computer hardware in both the public and the private sector. However, despite the increase of investments and progress within the field there are evidence to be found of a software crisis. This term was coined already in 1979 by the Controller General in a report to congress stating that "Contracting for Computer Software Development- Serious Problems Require Management Attention to Avoid Wasting Additional Millions-" The report concluded that the Government got less than 2% of the total value of the contracts. The notion of a software crisis is not only confined to the military or federal sector. Within private sector organizations, and personal computer software development traces of the software crisis are also found.

The symptoms of a software crisis are connected to chronically occurring problems through the software development stage. The records show that projects are plagued by budget-overruns, delays and late deliveries. Several of the projects are scrapped and discontinued before completion, other projects fail to meet the deadlines by several months or to meet budget limits with often as much as 30% or above.

Quality and reliability is also an aspect that shows signs of strains during the production, leading to poor quality and user dissatisfaction. Maintenance and additional improvement after delivery follows as a necessity, proving to be an additionally costly affair for the contractor.

In response to these observed problems, several efforts have been made to solve and bring discipline to the field of software development. The process of creating software has two dimensions, technical and managerial. On the technical side there have been significant inroads to cope with the technical hurdles. Practical working tools to support improved software production are commonly available. The managerial side has not been focused on in the same manner. Therefore there is a growing concern that the most significant hurdles to software development lies in the management of software projects. The continuous presence of cost overruns, failure to deliver on time and poor quality strengthens this notion.

The need for an integrative model.

The objective of this study is to develop and test an integrative view on software development project management in order to enhance the overall understanding and insight into the process. Rather than focusing on the aspects of development as isolated and separate issues, this approach seeks to build a system dynamics model in order to identify the relationship between each issue. The argument for this approach lies in our failure to understand the process of development as a whole. There are several hundred variables that affect software development, and these variables are not independent of each other. They are inter-connected, and they influence one another dynamically in a complex relationship through the development stages. The software crisis stems from our lack of complete knowledge of this relationship, and the integrative model thus combines both management-type functions as well as the production-type functions. No one management subsystem can perform effectively without the other subsystems.

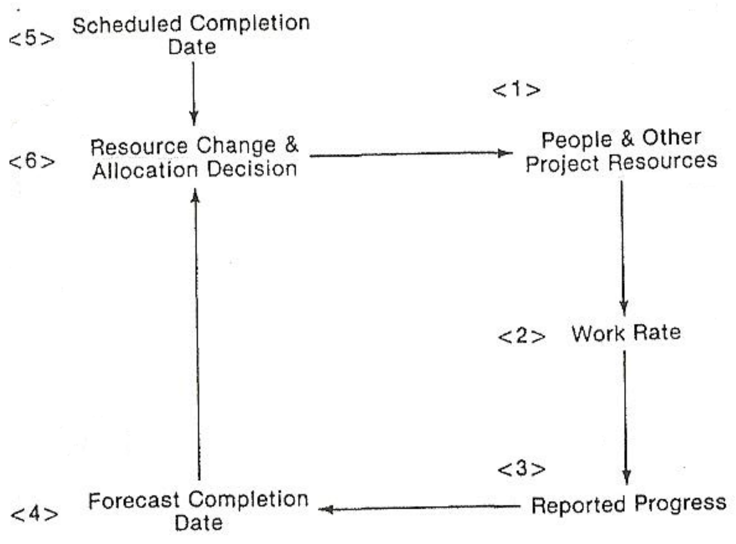
An integrative model will initially be able to shed light on the relationships between the subsectors, however the application of a system dynamics model will enhance our understanding even further. One of the core concepts of a System Dynamics model is the understanding of feedback. Feedback is the process in which an action taken by a person or thing will eventually affect that person or thing. The action and decisions made by a manager in one stage of the development process will therefore have an impact on the other stages of development through the entire software production structure. The software crisis indicate that managers are still not aware of how their decision-making on key strategical managerial aspects such as manpower-allocation early in a project will influence their ability to deliver on time and within the budget frame.

Summary Chapter 2

In this chapter the need for an increased understanding of the software development structure as a whole is further highlighted through the use of simple challenges facing any software project manager. In the previous chapter, one argued that the failure to address the managerial sides of a project directly cause misunderstandings and errors through the development process. When asked to model this relationship, one start with a simple mental picture. However, often these mental pictures are linear in nature, or fail to provide detailed understanding of the relationship between variables. The mental picture may provide relationships and concepts that are in fact false perceptions.

<u>The simple model</u>

The model depicted below is a simple model of the Software Development Process.



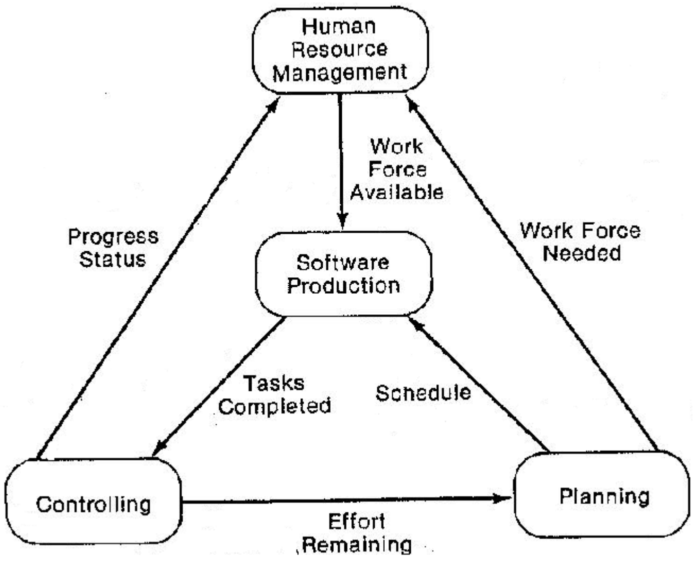
<5> Scheduled Completion Date
<1>
<6> Resource Change & Allocation Decision
People & Other Project Resources
<2> Work Rate
<3>
<4> Forecast Completion Date
Reported Progress

In itself this is a simple and tantalizing model that describes how project work is accomplished through the different stages. The allocation of manpower and resources provides the work rate. The work progress is then reported through a control system. The reported progress then leads to a forecasted completion date. The feedback loop is closed as the difference between the forecasted completion date and scheduled completion date causes adjustments allocation of resources to the project. This is a simplified model that easily portrays the process, however in response to challenges it holds clear weaknesses.

This model suggests for example that there is a direct relationship between adding people or resources and the work-rate. When more people are added the more work is accomplished. This would then suggest that if a project is running behind schedule the solution is to always allocate more resources and manpower. However, this ignores one vital aspect of the dynamic of software projects. When adding additional manpower to a project it leads to higher communication and training overheads. This in turn dilutes the team's overall production-rate, and the project falls behind schedule even more. This will delay the project even further. However, as the relationship is understood as increased workforce = increased work rate the manager's solution to the problem would always be to add additional people creating the same increase in training and communication overheads. This vicious cycle is often referred to as Brook's law which states "adding more people to a late software project makes it even later." This example illustrates clearly the need for a more detailed model, and that there are many variables both tangible and intangible that affect the software development process. These variables are not independent but interconnected in complex ways.

Holistic model

The solution to this problem is then to first build a holistic simplified model. This model is depicted in the figure below.



The model depicts the subsystems within a Software development project. The four major subsystems are Human Resource Management, Software Production, Controlling and Planning. This triangular system show how each subsystem is related to one another through different aspects of the software production process. In itself this is a simplified depiction, but it is useful as a starting point for building a complete system dynamics model. Through the additional chapters, these subsections are opened step by step in order to give a full detailed investigation of the variables and the relationship between them. This provides the basis for building system dynamics models for each subsection. Ultimately this will enable a connection of all subsystems into a large model for the entire process as a whole. In turn this will provide the details needed for truly understanding the complexity of the process, and provide a basis for testing management policies.

3.0 Model boundaries.

There are some notable boundaries to the model. First it only focuses on the developmental phases of software production, extending only to the last phase of software development. This excludes maintenance activities post development. Secondly the model excludes the requirements definition phase. In practice this approach assumes that the requirements for the project have been defined between the contractor and the team of developers. This relationship is then held constant through the process. External changes will consequently not be addressed by the model. The result is an endogenous point of view, where the important actions and policies are formed as a result of behavior within the system rather than as a result of external factors. The model is also limited to medium ranged software projects, excluding both super large projects with huge timeframes and budgets and small one-programmer projects.

The various assumptions and propositions behind the model are supported by references to the literature described in chapter 3 and interviews in chapter 4. In the appendix the DYNAMO equations for each subsystem is provided.

Summary Chapter 3

The third chapter focuses on the relevant literature that defines the parameters for the model, and the challenges facing the software development industry.

In the first stages of building knowledge of the issue at hand, Abdel-Hamid discovered that the literature on the field of software development was lacking. Therefore, the starting point for the venture of building an integrative model was the doctoral dissertation of Professor Edward B. Robert. Roberts's dissertation focused on challenges and issues linked to research and development. In order to define problems in a young industry, one decided to look into an already established field that copes with constructing a design into a product. Therefore the R&D sector was an obvious choice. Gehring and Pooch noted that many of the problems with poor accuracy on deadlines and cost-overruns were shared by both software development and R&D. Wolverton's paper from 1974 argues that the general principles involved in pricing large R&D efforts apply to large software projects. Putnam's SLIM model for software cost estimation is also based on R&D, through the work of Peter Norden.

Robert's dissertation was valuable in producing some of the subsystems linked to the managerial aspect of development. There are here some striking similarities between the observed needs in R&D and the needs in software development. Roberts identify for example the challenges in project planning, management of human resources and the control of progress. All of these sectors are important in the model presented by Abdel-Hamid, and indeed the challenges highlighted by Roberts are concerns shared in the software development field.

Expanding on Robert's original work, Nay and Kelly transferred the issue from single-project to multi-project environments. In both Nay and Kelly's work the multi-project environment create a competition for company resources between the different parts of the projects. The focus in both these studies was project lifecycle behavior. In contrast Edelman's work based on Nay's model look at the allocation of manpower resources and the effect of the management system on overall efficiency. Richardson took a different approach and focused on the output of a system rather than the internal behavior. Resources and average product development time could indeed be used to calculate the needed allocation of resources.

The last and more recent models forming the basis of the integrated model is the role of rework in project management. Several reports including reports from the US navy estimated large costs-overruns due to undetected errors and design-flaws in the final products. This observation was shared in both the software development aspect and the R&D aspect of military development. Rework is also focused on in Robert's R&D Model.

In order to structure a framework for the model, Abdel-Hamid needed to build on previous attempts to structure and formalize the subsectors provided in the final model. Richard Thayer's PhD dissertation of 1979 provided a fitting starting point. Thayer's work is the first recorded attempt at creating a complete model of a software engineering management system. The dissertation held two goals, (1) to develop and verify a generalized descriptive software project management model and (2) to identify and verify the key central issues in a software engineering project. Even though Thayer set out to create a model, it was the specific issues and problems identified that became the most valuable contribution to the study of software engineering.

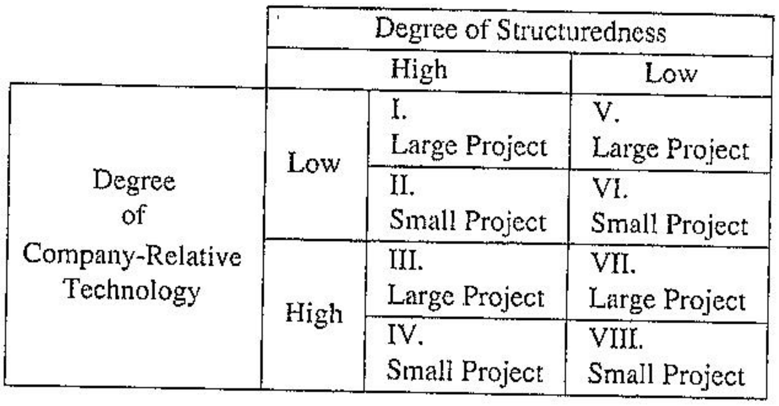
Thayer fist reviewed the existing literature identifying software engineering problems. Then by using his "software engineering delivery and success model", hypothesized which problems would affect the success of software delivery. He then categorized them on the basis of the classic management model. To verify his assertions, Thayer deviced two rounds of surveys. In the first round he conducted an opinion-survey with a selected group of members of the computer community. Technical leaders, software engineer authors, project managers, R&D personnel and software engineer educators were asked to classify the hypothesized problems as either "critical", "important", "not important" or "not problematic at all". From the 294 replies collected, 13 issues were marked as significant challenges, as they were labeled "critical" or "important" by at least 70% of the respondents.

The second round of surveys was sent to the project managers within the aerospace industry. By comparing the labels given in the second round with the labels provided by the first round Thayer could identify similar challenges between the two different fields of projects. Six challenges were marked as critical and common between both fields.

Riehl followed in the vein of Thayer and developed a planning and control framework to assist in the management of computer based information systems and development. His doctoral thesis compiled an extensive body of literature in order to identify concepts and practices deployed by authorities in the field of computer based information systems and electronic data processing management. From the gathered material he could attempt at a determination of policies and procedures actually employed in practice within the industry. His model was termed the "Composite-Working Model" which consists of 25 principles and

50 issues. The principles were the common policies and practices deployed through the literature, while the 50 issues were principles and policies deployed but argued over in the literature.

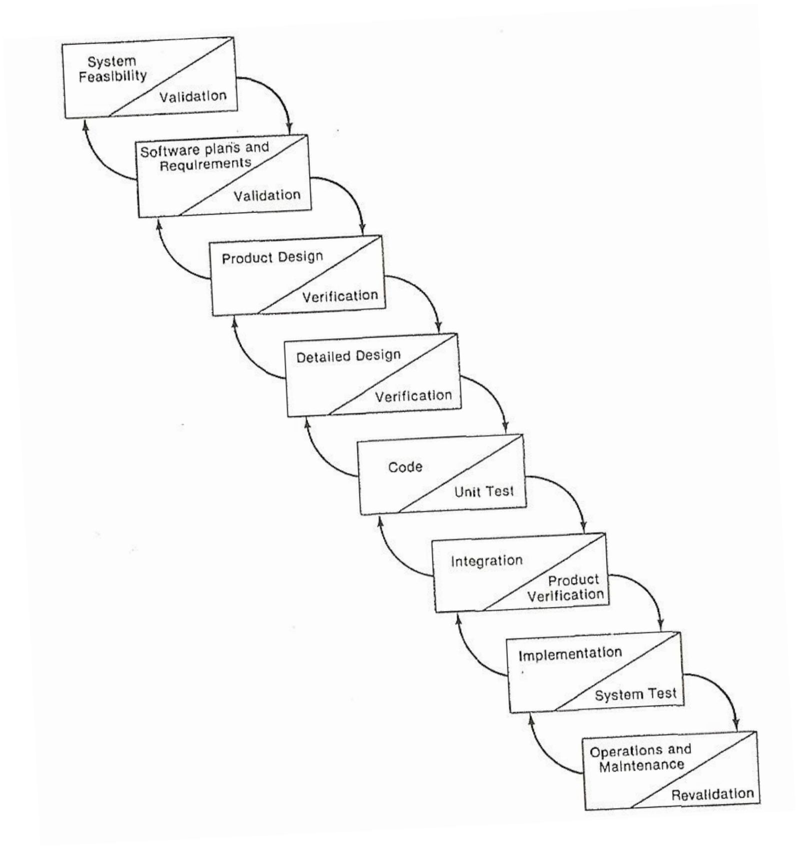
McFarlan's contribution differed from that of Riehl and Thayer by focusing on the differences rather than the common problems between different projects. McFarlan recognized that size of the project, organizational structure and the complexity of hardware deployed in the software project played an important role in determining what challenges manifested themselves in different projects. Thus his contribution provided a four-field table were 8 categories differentiated software projects across "Degree of structuredness" and "Degree of Company-Relative Technology"



		Degree of Structuredness	
		High	Low
Degree of Company-Relative Technology	Low	I. Large Project	V. Large Project
		II. Small Project	VI. Small Project
	High	III. Large Project	VII. Large Project
		IV. Small Project	VIII. Small Project

The concept Life-cycle

At a higher level of specificity there are research-efforts on the difference among phases *within* the life-cycle of a single project rather than between several projects. McKeen argued that the dominant organizing framework for application of system development is the concept of the life-cycle. The total developmental effort is apportioned into identifiable stages, were each stage represents a distinct activity characterized by a starting point and an ending point with deliverables in concert with an express purpose. The life-cycle model was formally acknowledged as an important element in systems development by its inclusion to the information system curriculum. Davis describes the contribution of the life-cycle concept to systems development as the recognition that all application systems must undergo the similar process when they are conceived, developed and implemented. If one part of the cycle is neglected it influences the entire system as a whole. The life-cycle has been modeled in different ways, but Boehm’s waterfall model is perhaps the most utilized in text-books.



Through the extensive body of literature on the software field, several different areas were identified as important in understanding the software project industry. Planning and the importance of planning were identified by scholars such as Davis, Zelkowitz and Shaw. Different modes of simulation and tests by scholars such as Thayer identified difficulties of planning. The challenges to estimation and the difficulty of planning human efficiency over time led to several theoretical and empirical quantitative estimation models. Putnam’s classical model is highly theoretical, the Watson and Felix model and Boehm’s COCOMO model are empirically based.

Human resources

In recent years people and organizational issues have gained recognition. With the personnel costs skyrocketing relative to hardware costs, the chronic problems in software development are more frequently traced to personnel shortcomings. Some are individual factors while others are group factors.

Motivation is a major individually based challenge to managers, and there are several factors linked to the motivation of employees. There are a few studies that focus directly on motivation of workforce, and they all conclude that work, achievement and growth are all important factors. Fitzenz's study provides 16 key areas of motivation. Among the 16 we find elements like "achievement", "possibility of growth", "recognition" "advancement" and "responsibility".

Other important individual factors are how employees are selected, for example through the IBM aptitude-tests, how the overall performance is measured and the rate of turnover. The turnover rate is highly linked to motivation, but also aptitude and professionalism play a role. This is noted by Bartol in his study on turnovers.

Group Dimensions

The literature on software development also features group dimensions to explain challenges and issues in software projects. These dimensions explain structural factors and process factors, and how these have an impact on the process. The structural factors explain how a project should be organized. There are three basic forms: Functional form, matrix form and project form. Youker suggests that these three organizational forms may be represented as a continuum ranging from functional at one end, to project at the other end. In between there is the matrix form, which includes a wide variety of structures both weak and strong. Weak near functional and strong near project. Several authors have presented proposed checklists for choosing the appropriate form for a project.

The process factors focus on the processes between members of a programming team. The most cited reference on the software development issue is found in this category of the field. Brooks suggests that human communication in a software development project is the most significant cause of slowdowns and obstacles. He also suggests that overhead results from training and communication. Each worker must be trained in the technology, the goals of the effort, the overall strategy and the plan of work. Training cannot be partitioned, so the amount of training effort needed varies linearly with the number of workers. The implication is that adding manpower to a project will increase productivity only to a certain point. Above this point communication and training delay will dominate the project. Hence Brook’s law state that “Adding manpower to a late software project makes it later” The relationship between human communication and programmer productivity was further studied by Scott and Simmons. They arrived at 8 consensus variables which have an important influence on productivity.

Control

When a plan has been put into operation, control is a necessary measure for progress. In order to identify deviations to the plan, and to apply the correct action control is paramount. However, in the literature there are several key observations that indicate control is a perilous activity. One observes for example that it is difficult to measure performance in programming as there is no physical product. One US Government spokesman called software programmers the “weavers of the emperor’s new clothes”. The manifestation of poor software project control manifests itself in several forms. For example in the “90% syndrome”, the production of inadequate software, systems that are inordinately expensive and the lack of a historical cost database.

The literature identifies two classes of explanations to why control is difficult in software projects; Product type factors and People type factors. Product type factors covers issues like the intangibility of the product during most of the development process. That the development cycle is highly complex, and managers fail to understand the complexity. Software system modules are not visibly linked to one another so changes in one module may have hidden or latent impacts on other modules.

People type factors focus on errors or misperceptions made by managers and programmers alike. One problem is the “wizard syndrome”. Management sometimes abdicates responsibility to highly trusted software specialists, rather than managing the system they put faith in the “wizard’s” ability. A second problem is inaccurate reporting, were programmers report inaccurate feedback regarding work-efficiency and progress rates. Often written records are “doctored” to hide delays or embarrassing setbacks. The last people-type problem mentioned here is the problem of optimism. Programmers are in general overoptimistic about their progress. Hence they underestimate the challenges ahead regarding production time and quality of their performance.

In the literature there have been many approaches to the issue of control. There have been attempts to create control standards such as Formal reviews, WBS, Automated Project Management System, PERT and GANTT. These techniques derive from other industries, and are linked into software development projects. However, the literature suggests that control and control standards may in fact lower the performance rates. Lehman’s survey surprisingly discovered that 17% of the software development projects in the aerospace industry had no control mechanisms. These projects fared better in regards to average on-time delivery compared to controlled projects. This finding was supported by Powers and Dickson in their MIS projects study. Bauer articulates this conundrum as: *“We are able to identify the sources of our troubles, but in many cases we have nothing to offer but good advice. We are in the situation of a physician who keeps trying out different pills on his patient in the hope that some will finally cure him”*

Summary Chapter 4

<u>Information resrouces</u>

In order to build the model of software project management, three steps of information gathering was conducted. These three steps were all fundamental in the creation and refinement of the system dynamics model, and provided the framework for both empirical evidence and the literature review.

<u>First round of interviews</u>

The first step was a series of interviews conducted with software development managers in several organizations. The purpose was to provide first-hand accounts of how software projects are currently managed in these organizations. A series of 10 interviews were conducted with software managers at DEC, MITRE and SofTech. The method utilized in these interviews is a technique called focused interviews.

Focused interviews are utilized in order to familiarize the interviewer with a given issue or field. The interviewers know in advance what topics to, or what aspects of a question they wish to cover. The list constitutes the framework of topics to be covered and they are derived from the research question. The manner of which the questions are asked, their formulation and when they are asked in the interview is left to the discretion of the interviewer. Such interviews are useful to obtain clear understanding of an issue and what to investigate further.

Literature review

By utilizing the initial integrated model as a roadmap, relevant literature for the study was systematically organized. This proved to be an extremely valuable tool, as the findings prompted the team to look into other relevant fields for ideas.

Second round of interviews

The third and final step was a second series of interviews to refine the model. It was exposed to criticism and scrutiny, revised and exposed again as long as the iterative process continued to be useful. This process refined the model and successfully increased the understanding of the problem for the participants.

In the second round of interviews there were a new set of interviewees, excluding those from the first step. In addition, seven interviews were added to the total expanding it from 10 to 17 interviews. This second round answered previous unanswered questions from the first round of interviews, and it provided higher accuracy to the model.

Summary Chapter 5

1.0 Human resource management

The human resource management subsector includes hiring, training, assimilation and transferring a project's human resources. The model below depicts the Human resource management subsystem.

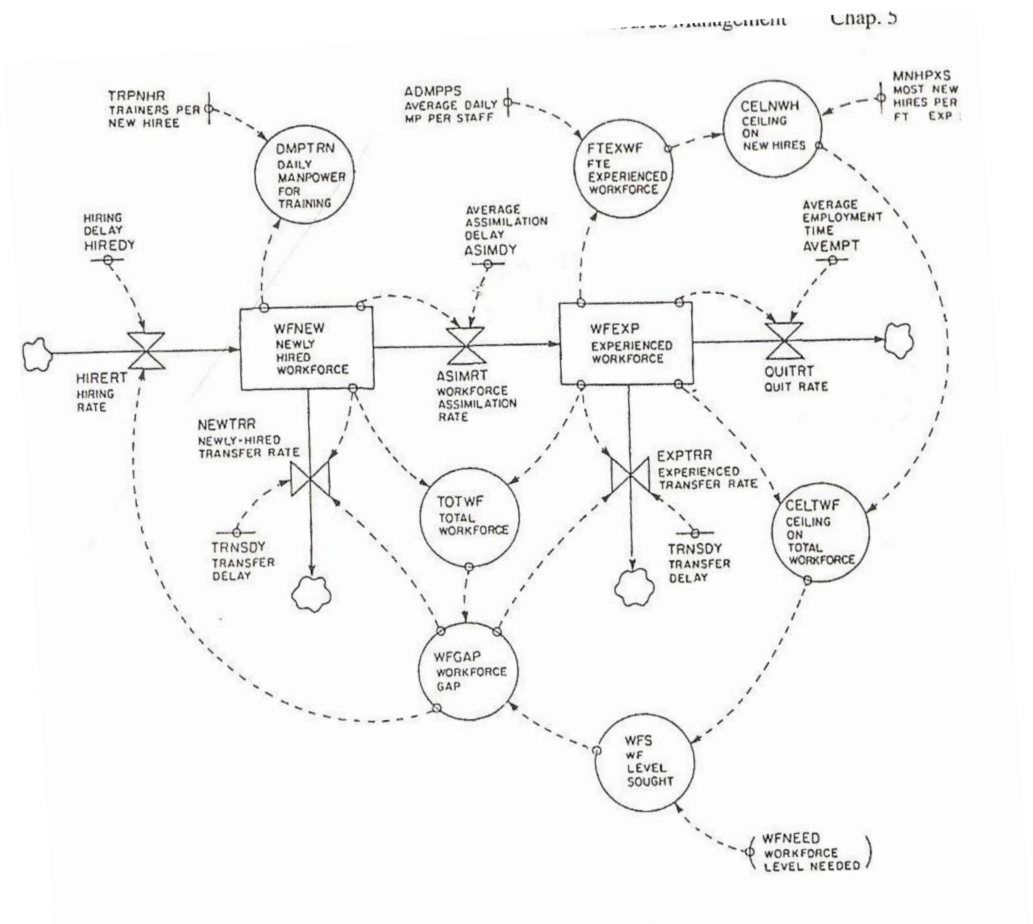

TRPNHR TRAINERS PER NEW HIREE
DMPTRN DAILY MANPOWER FOR TRAINING
ADMPPS AVERAGE DAILY MP PER STAFF
FTEXWF FTE EXPERIENCED WORKFORCE
CELNWH CEILING ON NEW HIRES
MNHPXS MOST NEW HIRES PER FT EXP
HIRING DELAY HIREDY
AVERAGE ASSIMILATION DELAY ASIMDY
AVERAGE EMPLOYMENT TIME AVEMPT
WFNEW NEWLY HIRED WORKFORCE
WFEXP EXPERIENCED WORKFORCE
HIRERT HIRING RATE
ASIMRT WORKFORCE ASSIMILATION RATE
QUITRT QUIT RATE
NEWTRR NEWLY-HIRED TRANSFER RATE
EXPTRR EXPERIENCED TRANSFER RATE
TOTWF TOTAL WORKFORCE
CELTWF CEILING ON TOTAL WORKFORCE
TRNSDY TRANSFER DELAY
TRNSDY TRANSFER DELAY
WFGAP WORKFORCE GAP
WFS WF LEVEL SOUGHT
WFNEED WORKFORCE LEVEL NEEDED

Figure 5.1 Human Resource Management Subsystem

The figure above shows that the project's total work force is assumed to consist of two levels. The workforce is differentiated into "Newly Hired Work Force" and "Experienced Workforce" There are two main reasons for this separation. Newly hired project members pass through an orientation during which they are less than fully productive. In this orientation the members are subjected to both the technical dimension and the social dimension of the project. On the technical side new recruits have to be trained and made familiar with the hardware, software packages, programming techniques and methodology of the project. On the social dimension, the new recruits must learn how to adapt and get along in the organization. For example they must accumulate norms and codes of conduct, understand the chain of command in the project structure, and how to best perform their role as a cog in the machine. Therefore the Newly Hired Work Force is on average less productive than Experienced Workforce.

The second reason for the differentiation lies in the relationship between them. The training of new recruits is done by members of the Experienced Workforce, thus a portion of the

Experienced Workforce is tied up in training new recruits. This leads them to be less productive as some of their time is spent on training. The time it takes for a new member of the project to be fully assimilated is on average between 2 and 6 months, and is captured as the flow rate between the Newly Hired Workforce and Experienced Workforce.

2.0 Determining Workforce Level

In order to determine the total workforce level, project management must consider many factors. One of the key factors is the current schedule completion date. In the planning phase management determines the work force level that is necessary to complete within schedule time based on the perceived remaining tasks. In addition, management assesses the stability of the workforce. Before hiring new members, managers try to estimate the time period new members are needed. It will take additional time and resources to add new members to the project as they will require training and assimilation before operating at peak efficiency. Therefore the work force level needed is not automatically translated into the hiring goal for human resource management. The ability to absorb new people is hinging on the assimilation-rate, and it claims resources from the experienced workforce that are now assigned to training new recruits. This is expressed in the model as the "Ceiling on New Hirees" which is equal to the "Full-Time-Equivalent Experienced Workforce" multiplied with the number of new recruits a full time staff can be expected to handle effectively. Management must carefully assess the gains of adding new members to the project contra the loss in efficiency of their Experienced Workforce plus loss of workforce stability.

Three factors thus affect management's determination of "Workforce Level Sought"; Schedule completion, work force stability and training requirements. "The Workforce Level Sought", and the "Total Work Force Level" then expresses the desired and actual workforce level. If there is a gap between the two, members will be hired or fired in order to equalize the difference.

3.0 Turnover

Finally the turnover rate affects the workforce. Turnovers are expressed as the quit-rate of the experienced work force. During the project some members will quit their job before completion. There are fluctuations in the overall quit rates, but in some years the annual quit rate in the DP field was as high as 25%. For every turnover, one experienced member must be replaced by a new member who needs assimilation and training. If mismanaged, this can cause significant delay.

Summary Chapter 6

The following subsystems all belong to the software production phase and are the major activities of a software development project. Development, quality assurance, rework and system testing. This subsystem is alone too complex to explain as one piece, therefor it is broken into four subsectors: Manpower Allocation, Software Development, Quality Assurance and Rework and Systems Testing.

Manpower Allocation

In the manpower allocation sector, management allocates their human manpower to the different subcategories of development. The total daily manpower available for a project is a function of the total workforce level and the average daily manpower per staff.

During the software production phase there are two main areas that management must allocate their resources between. The actual production of the software, which normally contains the main bulk of the total staff allocated, and the quality assurance and rework process. The challenge for management is to allocate the correct ratio between these two sectors in order to utilize their manpower efficiently.

Quality Assurance

Quality assurance is defined as a set of tasks performed in conjunction with the development of a software product in order to assure that it will meet specified standards. This is done through several techniques including walk-throughs, reviews, inspections, code reading and integration-testing. Hence, part of the overall production team is not involved in the actual software production but allocated to oversee the quality aspects of the product.

The ratio of allocated staff to quality assurance is however not fixed through the entire software production cycle. There are several variables that may increase or decrease the overall allocation of manpower to quality assurance. One of the main observed impacts is schedule pressure. When schedule pressure is high, the focus from management is mainly on the production side allocating most resources to actual production. The result is a reduction in quality assurance allocation, where some activities are suspended all together. Walk-throughs and Inspections are typical activities that are neglected when schedule pressure is high. This can be traced through the model as the gap between the actual fraction of manpower allocated to QA activities and the planned fraction of manpower allocated to QA. When schedule pressure is low, the gap is small. When schedule pressure increases the gap widens over time.

Rework

Through the QA process software errors are detected. Management must then assign manpower to correct these errors. The effort allocated to the rework sector is depending on the rework job that needs to be done, and the perceived rework productivity. There are often set goals on the desired amount of rework done per day, expressed in the model as "Desired Error Correction Rate". Errors are typically not corrected straight away, as they cause a delay in the software production process. When manpower is assigned to correcting errors, they are diverted from production or QA. When manpower has been allocated for QA and rework activities, the main bulk of the total workforce is then assigned to production.

Summary Chapter 7

The software development process consists of the design and coding of the software product. A software project is defined as a number of tasks, and the production rate is the number of tasks done per day. After allocating manpower to training, QA and rework the remaining bulk of the available manpower is assigned to the development of the product. This allocation continues until it is perceived that most of the software development tasks are completed, and the project is moved to testing.

During the software development phase, it is vital for management to conceive and predict the rate at which software is developed. Through the previous chapters we saw how allocation and pressures influence the staff's overall performance-rates. The development rate hinges on several factors and is therefore more complex than just being a function of how much manpower one allocates. In this chapter we differentiate between potential productivity and actual productivity. Actual productivity is expressed as the potential productivity subtracted losses due to faulty process.

2.0 Potential productivity

Potential productivity is defined as the maximum level productivity can occur when an individual or group uses their resources to best meet the task at hand. It is the level of productivity that can only be achieved when the individual or group work at peak efficiency. When there is no loss of motivation or faulty processes, the production rate is at its potential peak.

The nominal productivity rate is then the maximum level of productivity measured in the amount of tasks performed per day. This rate hinges on a constant factor, namely the density of tasks. In some projects the tasks are more dense and complex, and thus fewer tasks are conceivably done per day compared to a project with less dense tasks.

Experience and learning has an impact on the potential productivity. In chapter five the workforce was differentiated between newly hired workers and experienced workers. Experienced workers have a higher production rate than newly hired personnel. Therefore, on average, an experienced worker will have a larger potential output than a new worker. Hence, the ratio between new workers and experienced workers in the allocated workforce to software production will influence the potential productivity rate.

Learning also increases the potential productivity rate. As the project proceeds, the workers learn to do their job more efficiently. There is a learning-curve where the workforce over time increases their productivity due to being more familiar to the tasks required and how to solve them. There has been suggested that during the learning-experience, productivity for the worker involved can increase to as much as 25%.

3.0 Actual productivity

Potential productivity is the level of productivity an individual or group can attain when working at optimal rate with the best possible use of their resources. However, due to different losses caused by either communication or motivation the actual accomplishments of the group or individual rarely equals the potential. In a perfect world there would be no gap between potential and actual productivity.

3.1 Motivational factors

Motivational factors are linked to personal observations and goals during the software process. Some of these factors remain constant through a single project, while others change during production. Motivational factors like possibility for growth, responsibility, salary, advancement etc. are all constant factors that tend to characterize the overall climate of the organization as a whole.

Slack time is a motivational mechanism that changes over time in accordance with schedule pressures. Slack time is defined as the time lost to non-project activities. These activities include coffee-breaks, email reading, and personal activities. These are calculated as losses in man-hours. During the production process, some hours will be lost to these activities and will push actual production away from potential production. Positive schedule pressures however directly effects the room for slack time. When production falls behind schedule, the workforce is submitted to increased schedule pressure. Research shows that workers tend to cut their slack time, and devote more concentration to the work at hand when the pressure increases. Workers increase their man-hours in order to bring the project back on schedule. For a while this effect will increase the actual productivity as workers not only work harder during working hours, but also log overtime hours in order to close the time-gap. In such a situation the actual productivity rate can exceed the potential rate as this rate excludes overtime.

The positive effects on productivity is however limited. With increased schedule pressure there are over time adverse effects to overall productivity, as the above normal situation causes strain to the workforce. There seem to be an overwork threshold, and an exhaustion factor limiting the boost in productivity. According to the interviews, workers and managers are willing to increase their work-intensity within certain limits. If the need for overtime and increased labor-intensity is within these limits, there will be an overall boost from all workers in order to get the project back on track. However, if the need for overtime exceeds this threshold workers will only boost their productivity up to the threshold level and ask for an adjustment of the schedule instead. Exhaustion from the increased work pressure affects the overall productivity as well. When workers feel exhaustion from overwork pressures their production-rate decreases over time. After a period of over-work the workforce need time to reset and return to normal pressure levels. Slack time works as a psychological factor here, because it allows a break for the employees. They are enjoyable and remove the tension of the workload. When subjected to rising schedule pressures and reduced slack time workers feel deprived of such necessary breaks and the willingness to tolerate further levels of increased work pressure is lowered. For managers this poses a significant problem, as mismanaging these factors may ultimately lead to an increased quit rate and loss of experienced workers. As long as the willingness to overwork is zero, the extra man hours put in is equal to zero.

In some instances the opposite phenomenon occurs, where the project is ahead of schedule and it creates negative schedule pressure. In this situation there is an equal threshold for underwork as for overwork. During underwork situations, workers increase their slack-time in order to fill their days. However, if this situation persists workers will lose motivation over time and demand cuts in the schedule in order to create positive pressure.

3.2 Communication

The second factor of loss to actual production stems from communication. In a software development project, teams must communicate between each other in order to progress. For example, a designer must confer with a coder in order to answer any questions the coder may have on the design-aspects. Both designer and coder must then communicate with the individual testing the code in order to provide their experience with the program. These three must then talk to a documentor to assure that the documentation is proper and complete. This branched process of communication shows how communication is an overhead. With groups working on a project rather than just one individual, time is spent communicating between members which decreases the production-rate. This is defined as the average drop in a team member’s average nominal productivity as a result of team communication.

Summary Chapter 8

1.0 Quality assurance and rework

The development of software systems involves a series of production activities where the possibility for interjection and human fallibilities are substantial. Errors may occur at the beginning of the process where the objectives of the software system has yet been clearly or erroneously specified as well as during the production process when set objectives are mechanized. It is vital that the finished product performs to the functional standards intended by the architects, with as little faults or errors as possible. To achieve this standard of quality, quality assurance is carried out in tandem with the software production phase. Software quality assurance contains two distinct yet complementary methods. The first method is to design a coherent and complete set of requirements. The second is to review and test the product. In this model it is the latter method that is utilized, as the focal point is production.

Errors are generally understood to contain several different categories depending on error type. However, as this model focus on the management of software development, errors will not be put under different subcategories. They are instead implicitly captured by the model through behavioral differences. The way errors are generated and detected will vary through the development lifecycle.

Two sets of factors influence the error generation rate. The first set of factors includes organizational factors such as structured techniques, quality of staff and project factors. Even though these factors may differ from project to project they remain invariant during a single project. The cumulative effects of such factors are captured as the nominal number of errors committed per task. This rate is simply the product of the software development rate and the nominal number of errors committed per task. Since the error types vary over time the variable is not formulated as a constant. Typically design errors are found at the earlier stages of production, while coding errors are most prominent when the product progress towards completion. Design errors are costly and require more effort to detect than coding errors.

2.0 Error generation

There are several elements that influence the rate of errors made during the software production phase. The nominal estimation represents the effects of organization and project.

However, as identified in the previous chapter workforce mix and schedule pressure play a dynamic role during the software development phase.

The workforce was divided into newly hired workers and experienced workers. Newly hired workers are not only less productive than their experienced counterparts, but also more error prone. When the workforce consists of an experienced staff the error rate is close to the nominal rate as experienced workers tend to avoid making errors. The newly hired workers however are here assumed to be twice as accident prone. The mix of the workforce thus influence the errors generated. If the workforce mainly consists of newly hired personnel, the rate of errors generated will increase.

The schedule pressure also bears an impact on the generation of errors. During positive schedule pressures, workers will concentrate their efforts and put in more hours. However, exhaustion and stress increases the error rate. When programmers are tired, stressed or close to deadlines they make hurried decision that lead to more mistakes. With the added production rate they also produce more tasks per hour and may generate more errors per hour as a consequence.

3.0 Error Detection

Errors are potentially detectable as they are hidden until a task has been reviewed and tested. During this phase, some errors are identified and corrected through rework. However, errors are only potentially detectable and therefore some slip through and are detected at later stages. The detection of errors is therefore the essence of QA. This is done through several activities such as walk-throughs, reviews, code reading and inspections. There is a challenge to the QA activity that lies within this notion of errors. When QA is assigned, it normally follows pre-planned sessions. In the QA session window all tasks that have been completed are scrutinized and worked through, there is seldom any backlog. The reason for this lies in the difficulty in defining when a QA effort has been successful and should be finished. When a task has been worked through, and some errors have been detected there is no incentive to investigate further. In the mind of the participants the errors they have detected have been identified and sent for correction. It seems hardly necessary to reinvestigate tasks that have been worked through a QA session, however as the data show some errors slip undetected through the QA process.

As observed in the production subsystem, loss of motivation and communication also influences the potential detection rate. If the staff suffers from motivation and communication losses, twice the manpower is needed to allocate errors than under normal circumstances. If this is not picked up by the managers, errors will be able to move undetected through the QA effort and into later stages of production.

The QA manpower needed to detect an error hinges on error type, work efficiency and error density. Decreasing the QA manpower decreases the numbers or errors that can be detected. Manpower allocations to QA are often modest, and therefore the potential rate of error detection is lower than the actual rate of errors. Hence some errors slip by to the next stages of the software production cycle or end up in the final product itself.

4.0 Rework

Errors that are detected in the QA processes are reworked. The rework rate depends on the allocation of manpower to rework, and the time needed to rework an error. The nominal rework manpower needed per error is, as in QA depending, on the type of errors. Design errors are more costly and harder to rework than coding errors. The actual rework rate takes into consideration the nominal rate, and losses due to motivation and communication. As with QA, an unmotivated staff loses their productivity and an error takes on average more time to fix. With losses of motivation and communication, bad fixes also becomes a more frequent problem. Error correction is paradoxily also prone to errors .

Summary Chapter 9

1.0 Systems testing

System testing is the final subsector of the Software production cycle. Errors that are not detected by the QA efforts or bad fixes due to faulty rework remain in the product until the testing phase. Undetected errors do not lie dormant until they are detected and corrected in the systems testing phase. This is highlighted in the model by constructing two separate processes, where the growth and density of errors in the end influence the testing phase itself.

2.0 Error density and regeneration

Errors that go undetected in the QA phase, produce cumulative effects and regenerate errors through the process of development. For example a design error in one task will cause several errors in subsequent tasks as a trickle-down effect. Designs are the blueprint for codes, and faulty blueprints give faulty code builds.

Errors are divided into active and passive errors. Passive errors are dormant until detected in the testing phase, while active errors continue to create subsequent errors. Its dynamic is a system that feeds of itself, causing error-growth and error reproduction. All design errors are treated as active errors. In many ways the error regeneration rate depicts that of bacterial growth, multiplying and dividing over time creating more errors to handle. This is fundamental to the understanding of managing errors. Active errors left unchecked will multiply, giving a situation where one error needing correction has multiplied into ten.

3.0 System testing activities

When a project reaches its last stage of development, manpower is allocated from software production to testing. The objective of systems testing is to verify that all components of the product mesh properly, and that the overall function and performances are achieved. The manpower transferred from development to testing is gradual, but in the end the entire available workforce is allocated to testing. The objective of testing is then to test all tasks that have been developed, and remove any remaining passive or active errors.

The overall effort needed to testing and error-correction stems from the error density and the nominal testing manpower needed per error. Error density is derived from the sum of passive and active errors divided by remaining tasks to test. This represents the average number of errors per task. The effort needed to the testing phase per task is not only hinged on the error density rate, but also on efficiency of the workforce. As with production efforts and QA efforts, losses to communication and motivation reduces the efficiency and reliability of the testing phase which in turn may require more man-hours per task in order to fully complete the testing phase causing delays or increased costs.

The testing activity continues until all tasks that have been developed are tested. When testing is accomplished, the project is declared completed since this analysis only extends to the end of the testing phase.

Summary Chapter 10

Controlling

The control function in a software development project comprises of three elements. First element is measurement, detection of what is happening in the activity that is currently being controlled. This leads to the second element, evaluation. This entails an assessment of the significance of the activity being controlled. This is usually done by comparing information on how the activity is progressing with a preset standard or expectation on how the activity should be progressing. The last element is then communication. To report the measures and assessments so that behavior can be altered if the need is apparent.

In a software project, progress is measured by the number of resources consumed, or tasks completed, or indeed both. Measurement calculates the number of man-days still needed to complete the project. These include production time, time for developing QA tasks, to rework errors and test the software. This measurement is then compared with the actual man-days remaining before the deadline. If there are more man-days remaining in the calculation than actual man-days available the project is behind schedule. This assessment in turn leads to communication, where one solution can be to motivate the staff to work harder in order to get the project back on schedule. Other changes may be to extend the schedule itself.

Challenges to measurement of progress

The controlling phase of a software project is an ongoing process. From the planning stage, through the production stage and into the final testing stage, controls are carried out frequently in order to maintain progress estimations. However, this causes challenges to measuring progress.

The nature of a software product itself poses these challenges. In large parts of the development process a software product is an intangible entity. There are no visible milestones to measure progress from, as there is no physical product. It is therefore hard to distinguish progress from any produced output. In the interviews conducted from this study, the managers interviewed revealed that instead of creating milestones, progress was measured by the rate of which resources was expended. For a 100 man-days project, it would be estimated at 10% completion after 10 man-days had been expended. This way of measurement yields in reality no information, and the result is reports that mimic the estimated progress planned.

In the later stages of software development, the actual accomplishments and productivity of the work becomes more visible. This in turn leads project members to be more able to perceive how productive they have actually been. This in turn provides actual measurements for control on the progress of the project. The new and more accurate measurements may then reveal considerable discrepancies and adjustments are prompted. Due to the delay-factor, the density of adjustments enters towards the end of the development process.

Adjustments

When progressive measures are more readily available, and assessments can be made, adjustments are carried out. One of the adjustments may entail the project's job size. When there is a situation where the number of man-days still needed to complete the project exceeds the number of man-days available, one way of adjusting is to alter the scope of the project. In the chapter on software production we learned that the workforce itself can to some degree absorb man-day shortages through increased production due to schedule pressure and overwork. However, this absorption-rate is not unlimited. If the excess supersedes the ability to absorb, it is necessary to adjust the scope of the project in terms of man-days. For example a 100 man-day project may be redefined to a 115 man-day project. When reports through control reveal such factors, the project managers react to transform the revised perceptions about the job size into actual adjustments as the example above illustrates.

The same type of adjustments in scope and size is linked to tasks as well. The number of actual tasks needed compared to the estimated tasks needed, may lead to upgrading or downgrading a project's scope on task-size. This in turn will translate into man-days needed to finish the product.

Summary Chapter 11

<u>The planning phase</u>

In the simple integrated model the last part of the production triangle is the planning phase. Its position is misleading however, as the planning phase is the starting-point for any software project. Before production and control is carried out, the fundamental framework on the project's size, budget frame and scope is chiseled out. The estimated completion date (measured as man-days required), workforce density and allocation of staff to the different parts of the project is established.

During the process of production, these pre-established goals are the foundation for the control effort. Actual progress and development is compared to the planned perception of workforce needs, time for completion and job size. While the production advances through the different stages, assessments from control provide adjustments to the initial plan.

This is a key subsector for the managerial effort on the software project as a whole. Policies regarding working hours, work force density, job size, and completion date are considered and set here. These policies are then fed back to all the other subcategories in the model. Therefore planning-adjustments hinges on several different factors addressed in all the previous sectors. For example, if there is a difference between the indicated workforce needed for the project and the actual workforce, people may be hired or fired in order for the gap to be closed. However, as described in the chapter on human resources we see that hiring and firing of employees is not only based on the schedule considerations but also on workforce stability and the mix between new and experienced workers.

This subsystem then works as the information-hub of the entire system. The different considerations and issues observed during the project's lifecycle in the previous sub-sectors form the basis for decisions and adjustments to plan. This in turn leads to a complex and intertwined planning-structure were all the different considerations play out in management-policies. The time that is remaining for the scheduled project, the perceived number of tasks still remaining until completion and the need for adjustments in the workforce despite a desire for workforce stability are all considerations that interplay with determined policy. These implications and problems must be solved by the project managers through planning and revision to the plan.

Summary Chapter 12

A case study

The twelfth chapter focuses on the results from the case-study conducted at the Systems Development Section of NASA's Goddard Space Flight Center (GSFC). The objective of the case study is to examine the model's ability to reproduce the dynamic patterns of a completed software project. In the study the author tracks a set of variables pertaining to project management, including completion date estimates, man-day estimates, cost and work force loading. The case in question is an organization engaged in the development of application software that supports ground-based spacecraft altitude determination and control. The SDS was engaged in a suitable project named the DE-A project, which was utilized as a reference to the model. The model is used to simulate the software development for the DE-A, while the actual data from the project is used for comparison in order to establish the model's accuracy.

The DE-A project

The requirements for the DE-A project were to design, implement, and test a software system that would process telemetry data and provide definitive attitude determination as well as real-time attitude determination and control support for NASA's DE-A satellite.

The DE-A project was selected for the case study by NASA to satisfy three criteria furnished by the research-team: It had to be medium sized (16-64 KDSI), recent and "typical". Typical is here referring to a system developed in a familiar in-house software development environment.

The data presented from the case study was extracted from two sources. The first source is interviews with the Head of Systems Development Section of GSFC who managed the project. The second source of information is the project documentation itself. These documents include both the Software Development History for Dynamic Explorer Attitude Ground Support System, and the DE-A Resource Summary.

The lifecycle phases covered by the case study include design, coding and the systems testing phases. Excluded from the study are the requirements definition phase and the acceptance testing phase. These phases had not been included in the observed group's project responsibility.

The project started on 1st October 1979 and ended on the 24th of April 1981. The development and target operations machines were the IBM S/360-95 and -75. The code language was mostly Fortran (85%), while assembler language and assembler language macros constituted the remaining 15%. The size of the system in Delivered Source Instructions is 24,400 DSI.

Before the simulation could be run, the model's parameters had to be set. Four sets of parameters were set in order to simulate the DE-A project, and they are listed in the figure below.

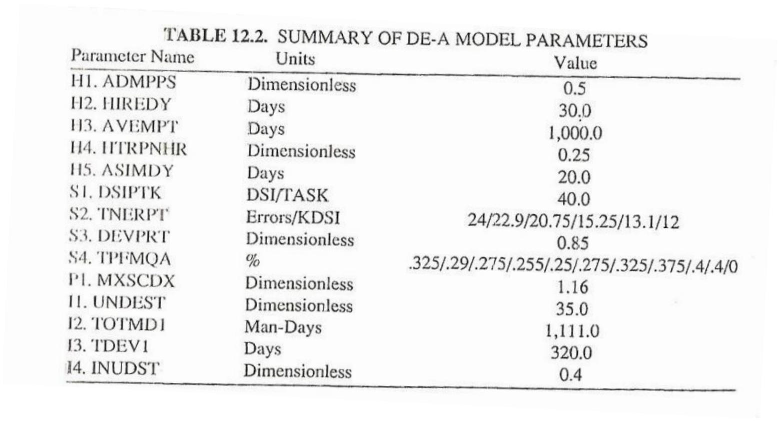


TABLE 12.2. SUMMARY OF DE-A MODEL PARAMETERS

Parameter Name	Units	Value
H1. ADMPPS	Dimensionless	0.5
H2. HIREDY	Days	30.0
H3. AVEMPT	Days	1,000.0
H4. HTRPNHR	Dimensionless	0.25
H5. ASIMDY	Days	20.0
S1. DSIPTK	DSI/TASK	40.0
S2. TNERPT	Errors/KDSI	24/22.9/20.75/15.25/13.1/12
S3. DEVPRT	Dimensionless	0.85
S4. TPFMQA	%	.325/.29/.275/.255/.25/.275/.325/.375/.4/.4/0
P1. MXSCDX	Dimensionless	1.16
I1. UNDEST	Dimensionless	35.0
I2. TOTMD1	Man-Days	1,111.0
I3. TDEV1	Days	320.0
I4. INUDST	Dimensionless	0.4

From the actual DE-A resource summary and the data provided from the interviews the research team was able to carefully calculate the parameters for each variable. The table above summarizes these parameters providing their DYNAMO variable tag, unit measure and value.

The process of parameterization did not involve any of the policy formulations. The set of parameters defines the particular environment within which policies are exercised. By setting parameters such as Hiring Delay and Turnover, one does not alter the rational that determines how hiring and firing decisions will be modulated through the life-cycle. Therefore one can determine the starting point of 1.5 full-time equivalent employees at the start of the project. However, it is not possible to ascertain the project's work force loading pattern. The dynamic behavior of management systems tends to be a function of the interaction of policies that govern such systems.

After the model's parameters had been set, it was run to simulate the DE-A project. This allowed us to perform a crucial test of the model's validity and accuracy, by performing a simulation of a true empirical test-case. For the integrative model to have any strength beyond being a theoretical simulation, it must be able to within certain limits reproduce empirical behavior in a reported software project.

Figure 12.3 shows the DE-A’s estimated completion date measured in terms of number of elapsed days until completion, and total work force expenditures measured in man-days charged during the project. The actual numbers reported in the DE-A report are shown as circles with a dot inside.

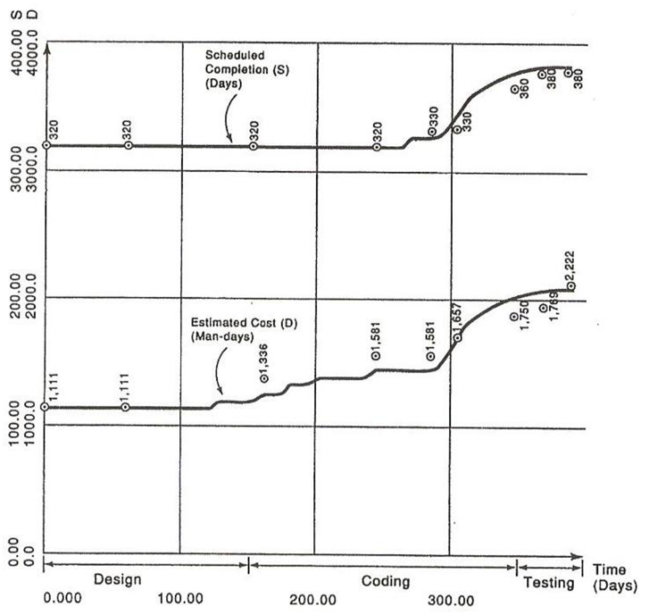


Figure 12.3 Scheduled completion and estimated cost (simulated and actual)

The model accurately portrays management’s hesitation to adjust the project’s scheduled completion date during most of the development phase. Adjustments are made instead to the project’s workforce level. This pattern arises according to DeMarco for political reasons, as original estimates are made, it is tempting to keep these estimates rather than re-measure them. Adjusting early in a project may give the impression of a slip-up, and if you’re new estimates prove to be wrong later you may appear to slip up twice. One tends to stick to the original estimates, even though they are substantially off target.

Towards the end of the design phase, the DE-A project’s man-days budget are adjusted. This is triggered as undiscovered job tasks are discovered. The project was initially underestimated, perceived to be only 16 KDSI. The adjustments made in the DE-A project are somewhat larger than those estimated by the model. This is an indication of greater visibility for managers in the project than assumed in the model. Hence, managers were able to detect discrepancies of the actual scope of the project earlier than in the model. However, even though the visibility of the project was higher than the industry norm, it is far from perfect. Significant adjustments were necessary to both the project's man-days and schedule during the software development process. This is an outcome that the model perfectly reproduces.

The model's values for the project's final man-day expenditures are slightly lower than the actual. The model calculated a total of 2092 days while the project actually ran for 2222 days. The reason for this discrepancy is that even though the model fully reproduces the project's manpower loading patter, it slightly underestimates the values of manpower level. Lower manpower levels mean lower communication and training overheads, which mean a slight over-estimation of productivity.

The model's project duration estimate was 387.5 days. This is slightly longer than the DE-A actual project duration of 380 days. The reason for this is the more aggressive management behavior of the DE-A managers than assumed by the model. As described earlier, due to the nature of this project there was a low tolerance for delays and errors due to a space-launch date. This behavior was strongly apparent at the final stages of the project, as more people were added. The work force level at the end of the system testing phase was 16 fulltime employees rather than 14.8 predicted by the model. With more people at hand the actual project a smaller schedule overshoot was achieved.

Manpower Loading

When it comes to man power loading the model accurately replicates the actual DE-A pattern. It reproduces the "a-typical" work force loading pattern. The "Typical" pattern discussed in the literature on software project work force patterns is a concave curve that rises, peaks and drops to lower levels as the project proceeds towards the end of system testing phase. The work force level shot upward towards the end of the project because of NASA's tight scheduling constraints. This is depicted in the graph below.

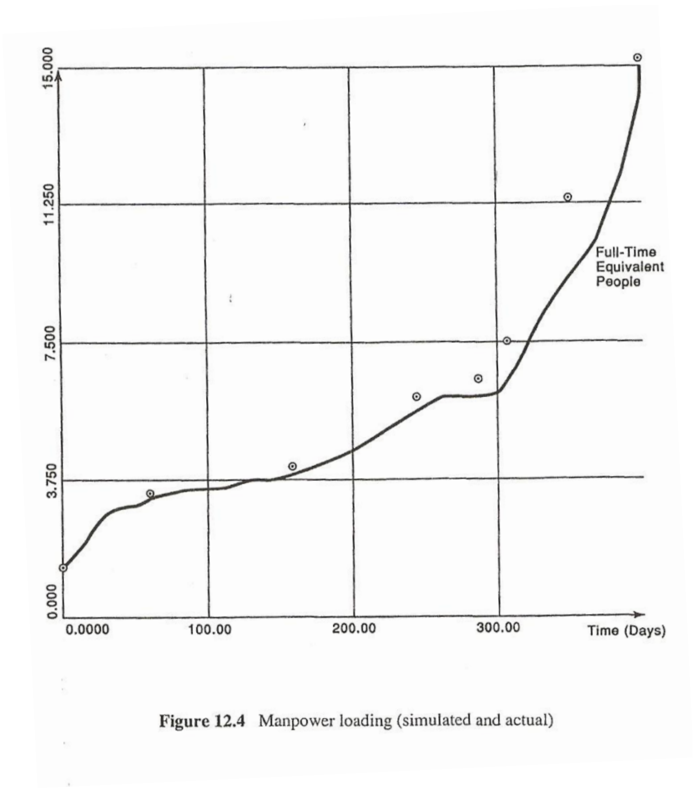


Figure 12.4 Manpower loading (simulated and actual)

Cumulative Man-Day Expenditures

In the final figure 12.5 the model's cumulative man-day expenditures are plotted together with the actual results. The model captures the exponentially increasing pattern. The actual figures are slightly higher, however, because of the earlier mentioned underestimation of the project's actual work force level. DE-A management's "Willingness to Change Workforce Level" did not decrease as the project proceeded towards the final stages as much as the model assumed.

Summary Chapter 13

Model Behavior

System dynamics modeling is a tool that enables us to conduct repeated experimentations with the system at hand. With a system dynamic model one can test assumptions or alter policies. The objective is to gain an understanding of the managerial policies available, and make predictions on their implications. Using the system dynamics model as a vehicle for experimentation can provide many benefits to the software engineering community. As Weiss observed: …in software engineering it is remarkably easy to propose hypotheses and remarkably difficult to test them…"

Unfortunately, controlled experiments in this field are difficult as they tend to be costly and time-consuming. Furthermore the isolation of the effect and the evaluation of impact given by any given practice within a large, complex and dynamic project environment can be exceedingly difficult. In addition to permitting less costly and less time consuming experimentation, simulation models make perfectly controlled experiments possible.

The EXAMPLE Software project

EXAMPLE is the prototype project for the experiments in this book. The model is run to in order to allow observations and analysis of the EXAMPLE behavioral patterns on several significant project variables. These include "Workforce Level", "Schedule Completion Time" "Errors" and "Productivity". The goal is to demonstrate the model's ability to replicate the reported behavior in the literature. The EXAMPLE parameters will be used in chapters 14 to 19 to test the implications of an array of managerial actions, policies and procedures. EXAMPLE is 64000 DSI project.

Setting Nominal Potential Productivity

The productivity was as demonstrated in chapter 7 not defined in terms of DSI/Man-Day but rather as Tasks/Man-Day. A "Task" is defined in terms of a number of DSI, and its value for a particular simulation is set to the numerical value of "Nominal Potential Productivity" that is expressed in DSI/Man-Day. For example if 50 DSI is the nominal potential productivity rate then the value of "Task" is 50 DSI. Thus the value for Nominal Potential Productivity will be 1Task/Man-Day. In order to determine the value of "Task" in EXAMPLE we therefore need to determine the Nominal Potential Productivity rate in terms of DSI/Man-Day and set the "Task" value accordingly.

Determining Actual Productivity

In order to determine the level of actual productivity, the author here turns to Boehm's book *Software Engineering Economics,* which provides ample data from the software development environment at TRW. To maintain consistency EXAMPLE will be characterized totally from the TRW data. EXAMPLE draws on projects described by Boehm as the most common type of software projects. Small to medium-sized projects developed in an in-house, familiar and organic software development environment. For a 64KDSI project, the TRW data indicate an overall productivity of 338.4 DSI/Man-Month. This value is arrived at by dividing the project's size in DSI by the total effort spent to development, QA, Rework and testing. The data also indicates that testing consumes approximately 22% of the total effort, while the QA activities range from 15 to 20% of the total effort. Note that the rework of errors during development is not given an explicit figure of the total effort.

The amount of effort expended on designing and coding the product is half the total man-days expended on the project. The development productivity is 2 x 338.4 = 676.8 DSI/Man-Month. This number is divided by 20 in order to translate it to man days. The actual productivity level is therefore 33.84 DSI/Man-Day. This is the number that reveals the actual productivity rate, but as explained in previous chapters, the production rate is influenced by motivation and overhead losses. To find the nominal productivity rate on have to determine the loss of motivation and overhead.

Determining Nominal Potential Productivity

Actual productivity rarely equals potential productivity because of the losses mentioned above. In the model actual productivity is the product of potential productivity and the "Multiplier to Productivity due to Communication and Motivation Losses". If the value of the multiplier can be estimated, one can divide it into the value of actual productivity to come up with an EXAMPLE estimate for "Nominal Potential Productivity".

The multiplier itself is the product of two variables: "Actual Fraction of Man-Days on Project" and "Communication Overhead". The nominal value of the former was set at 0.6, indicating that a full-time employee allocates on average 60% of his or her time to the project. The communication overhead is a function of the team's size. In a 64 KDSI project the average staffing level, according to the TRW, are 10 people.

From chapter 7, it was determined that the loss due to communication overheads is 6%. The multiplier is therefore calculated to be 0.6 X (1-0.06) = 0.564. By dividing this value with the actual productivity level we estimate the potential productivity to be 33.84/0.564 = 60 DSI.

<u>Initializing Schedule and Manpower</u>

In order to determine EXAMPLE's manpower and schedule allocation variables, Boehm's COCOMO model is utilized. The COCOMO model is a software project estimation model developed and used by the TRW. COCOMO exists in a hierarchy of detailed forms, and for this analysis the "Basic COCOMO" version was used. This version is deemed by Boehm to be best suited for EXAMPLE's type of project.

The development period covered by COCOMO estimates begins at the beginning of the product design phase and ends up at the end of the systems testing phase. The primary input is the perceived size of the project in KDSI. Perceived size, because the real size is often unknown in the beginning of a project.

Boehm argues that software under sizing is one of the most critical problems to accurate software cost estimations. This is substantiated through the literature on the field. This notion of under sizing must be translated into the model, but how much under sizing? For EXAMPLE the assumed underestimate of project's size by managers is a factor of 1.5. This value conforms to Boehm's estimates. A project's size of N KDSI would be incorrectly perceived as being only 0.67N KDSI. For EXAMPLE this translates into a situation where management will perceive the project's size to be only 42.88 KDSI rather than 64KDSI. This value becomes the input that management uses in their COCOMO's effort and schedule estimation values.

The COCOMO equation for the number of man-days to develop and test the project is:

$MD = 2.4 \times 19 \times KDSI^{1.05}$

EXAMPLE's equation: $MD = 2.4 \times 19 \times (42.88)^{1.05} = 2359$ man-days

This represents the total man-days to develop and test the software product. For planning purposes this effort is then distributed among the project's life cycles phases. For the question of allocation, Boehm provides several phase distribution guidelines. For a 42 KDSI project the TRW data suggests a development to testing distribution of 80% to 20%

MD for Development = 0.8 x 2359 = 1887 man-days

MD for Testing = 0.2 x 2359= 471 man-days

In addition to estimating the project's man-day requirements, management also estimates the project's development time and staffing level. The COCOMO equation for total development time is:

TDEV = 47.5 x $(MD/19)^{0.38}$ days

Substituting for the value of man-days the equation is

TDEV= 47.5 x $(2359/19)^{0.38}$ = 296 days

Finally the average staffing level is determined by dividing the estimated value of total man-days by the estimated value of the development time. For EXAMPLE we get:

ASL = MD/TDEV

= 2359/296 =8 full-time equivalent software personnel

We assume that project members of EXAMPLE will work full-time on this project. The model's parameter "Average Daily Manpower per Staff" would be set to 1 man-day. The average staffing level calculated above would be 8 actual software personnel. Not all 8 personnel will be on-board at the beginning of the project. Most software projects starts with a smaller staff of core designers, and as the project grows in size the staff grows with it. For EXAMPLE we assume that the project starts with half of the ASL equal to 4 software personnel. This is a consistent figure with the results reported in Boehm's book.

With these parameters determined EXAMPLE can be run to simulate and observe its behavior. There are 3 main areas that will be discussed in this chapter: Project progress, Manpower Distribution and Work Intensity.

Project Progress

The dynamic behavior of six key measures of progress is depicted in the figure below. These six measures are: cumulative tasks developed, cumulative tasks tested cumulative man-days, the perceived job size in tasks, the perceived job size in man-days and scheduled completion date in days. The table provides key statistics for the project.

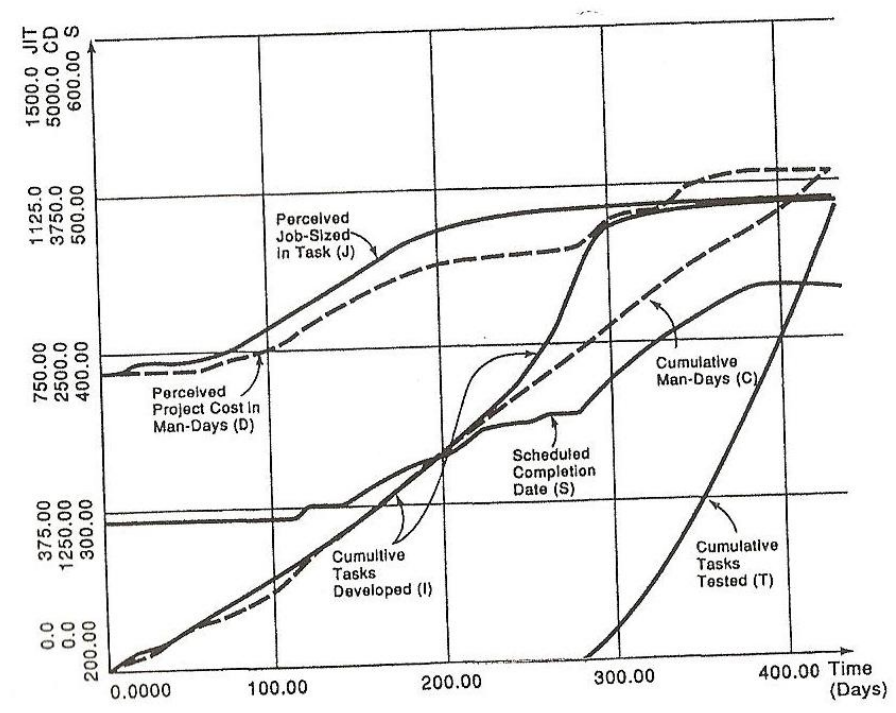


TABLE 13.1 KEY STATISTICS OF PROJECT EXAMPLE

1. Project Size			=	64,000	man-days
2. Man-Days					
	Total		=	3,795	man-days
	Development		=	2,681	man-days
		Coding + Design	=	1,782	man-days
		QA	=	380	man-days
		Rework	=	519	man-days
	Testing		=	1,114	man-days
3. Completion Time			=	430	working-days
4. Errors					
	Total Error Generated		=	1,494	→ 23 Error/KDSI
	Total Error Caught During Development		=	728	→ 49% of Error Generated

Discovery of Additional Job Tasks

EXAMPLE was perceived initially by management as a 42.88 KDSI project rather than a 64 KDSI. This underestimation leads management to believe that it comprises only of 746.6 tasks instead of 1067 tasks. As the project develops “Undiscovered Job Tasks” are progressively discovered as the true size and scope of the project becomes visible. The rate of which perceived job size rises remains low for a significant portion of the development phase, before it starts to accelerate rapidly. This behavior is consistent with the behavior observed in the NASA case study. The initial design phase of development constitutes the overall system designs. Many implementation details such as help message processing or error processing would still not be visible. When the project moves into the detailed design phase, these necessary jobs are discovered rapidly. As the additional tasks are discovered, and management starts to realize the project’s scope is larger than expected adjustments are made in the project plan to accommodate the additional workload. This is visible in the figure above, as “Job Size” and “Schedule Completion Date” are adjusted upwards. Two interesting observations are made at this stage: (1) the adjustments prove inadequate to fully accommodate the additional work, and (2) the first adjustments to the schedule lags considerably behind the first adjustment to man-days.

Adjustment to Man-Day Estimates

The additions to the project’s man-days and schedule that are triggered explicitly by the discovery of undiscovered tasks level off at approximately day 200, when most undiscovered tasks has been identified. In EXAMPLE we see that the job size levels off at 1067 tasks, the true size of the project. At this point the project’s size in man-days levels off at a value of 3200 man-days. However as depicted in figure 13.1, while perceived job size remains unchanged, significant changes are made to the project’s man-days and schedule. As management realizes at day 300 that the project is behind schedule the “Total Man-Days Reported Still Needed” to complete the project exceeds the “Man-Days Remaining” in the projects plan. In this case the man-day shortage problem is largely the result of the project’s under sizing problem. When additional tasks are discovered in a software project, the additions made in the project’-days to accommodate them are not sufficient. The reason is that some of the discovered tasks are absorbed by project members without any formal adjustment to the plans.

In EXAMPLE by day 200 when almost all job-tasks have been discovered, the project attains the true size of 1067 tasks. To this value, the “Job Size in Man-Days” would be raised to 3200 man-days. However, this would not be enough to accommodate all the additional tasks. If we use the COCOMO’s MD equation for a project perceived to be 1067 tasks from the beginning we find that:

$MD= 2.4 \times 19 \times (64)^{1.05}= 3593$

Increasing the number from 2359 to 3200 man-days, fall short of the 3593 man-days needed by the COCOMO estimation. This is a significant deficit in the project’s man-days budget.

This deficit in man-days is handled when it becomes visible. This happens towards the later stages of development, when the development work is nearly finished or the allocated man-days budget is consumed. Once visible the man-days deficit is handled by working overtime and/or one adjusts the project’s man-days budget upwards. For EXAMPLE both policies are employed.

Adjustments to Scheduled Completion Date.

The second adjustment for figure 13.1 concerns the scheduled completion date. The first adjustment to the schedule lags considerably behind the first adjustment to man-days. The adjustment to job size in man-days is made around day 80, while the adjustment to the schedule is made at day 140. The reason for this lag lies in management’s policy on how to balance work force and schedule adjustments throughout the project. In the early stages of a project, management is generally willing to adjust the work force level to maintain the scheduled course. However, as the project advances through the stages, management grows unwilling to hire new personnel to the project in order to maintain workforce stability. Hence the initial responses to newly discovered tasks that increase the project’s man-days are absorbed by adjustments to the workforce alone. Later in the project, management shifts focus from adjusting the workforce to adjusting the schedule. The figure below, show the workforce level through EXAMPLE's lifecycles.

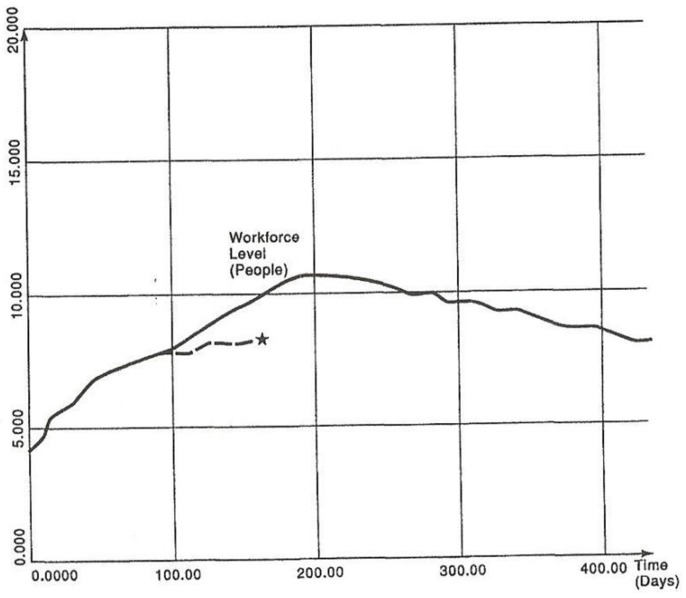


Figure 13.2 Workforce Level

Manpower distribution

The figure above depicts the shape of EXAMPLE's manpower-distribution curve. This curve conforms to the manpower distributions reported in the literature. It has the shape of a concave "wave", and compared with IBM's reported workforce level at DP services Organizations we find a striking resemblance. It is important to note that EXAMPLE was also able to reproduce the DE-A case study workforce load that was "a typical"

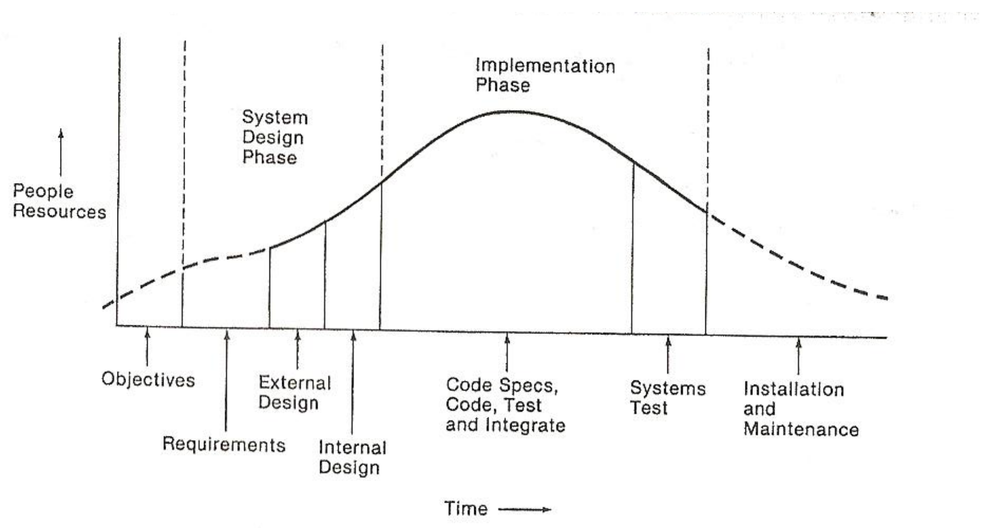


Figure 13.3 Workforce Level at IBM's DP Services Organization

Work Intensity

The typical 8-hour day of a full-time staff member on a software project is not entirely devoted to productive project-related work. As mentioned earlier only 60% of the time is on average spent on direct project-related work. The loss of productivity does not of course remain constant at the 40% level throughout the life of the project. Motivational effects of schedule may push the "Actual Fraction of a Man-Day on Project" to higher or lower values. Positive schedule pressure arises whenever the project is perceived to be behind schedule. In such a situation we argued earlier that software developers tend to work harder, allocating more man-hours to the project in an attempt to compensate and bring the project back on schedule.

Impact on Actual Fraction of Man-Day on Project.

The dynamic behavior of the "Actual Fraction of a Man-Day on Project" for EXAMPLE is illustrated in the figure below. The two spikes in overwork occur as an explicit project milestone is approached. The first spike occurs towards the end of the development phase, the second at the end of systems testing. This pattern has been observed by Boehm and labeled as the "Deadline Effect". In a simulation model such as EXAMPLE it is possible to isolate and combine the effects of suspected factors in order to pinpoint the mechanisms responsible. In our EXAMPLE case, the shortage of man-days is perceived late in the development phase around day 180. The reaction from the project member's is to work harder in an attempt to compensate. Working harder translates directly to the Actual Fraction Man-Day on project.

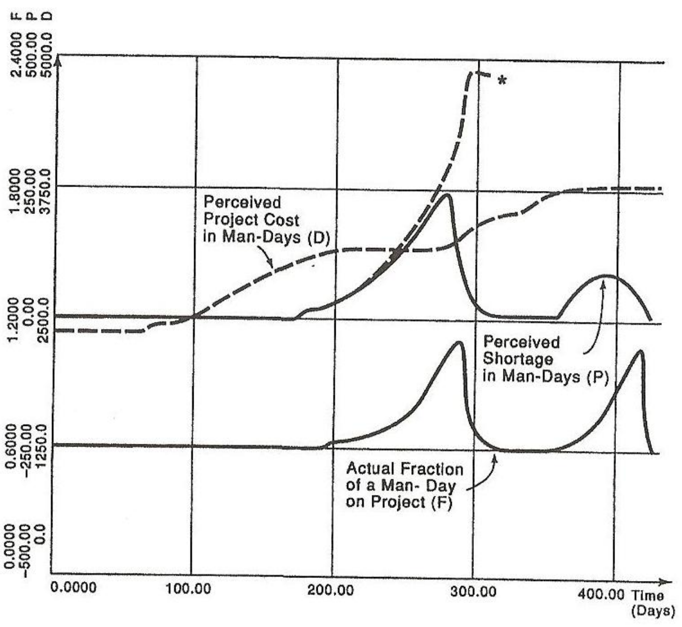


Figure 13.5 Work Intensity

Even though the project members are working harder, they are only cutting into the man-day shortage. The reason for this is the fact that as the project closes to completion, the actual size of the shortage become apparent. Thus the shortage of man-days increases steadily as the degree of visibility exposes even more shortages.

<u>Impact on Overwork Duration Threshold</u>

Project members are never willing to maintain an above-normal work rate indefinitely. Once people start working above their normal rate, their overwork duration threshold decreases. As mentioned in chapter 6, workers enjoy their “slack-time”. If “Slack-time” is suspended over time, the workforce grows weary. As this threshold decreases the maximum number of man-days of backlogged work that project members are willing to handle in addition to their planned work decreases as well. In EXAMPLE this is exactly what happens. The persistence of the man-day shortage exhausts the work force’s intensified efforts around day 300. In response to this, adjustments are made to handle the remaining shortages through the man-days budget and schedule.

The same sequence is observed at the testing phase. As testing progresses, the system’s error proneness becomes visible. As this happens, any shortages in man-days for the testing phase become more apparent. This spikes a second increase in project member’s efforts to bring the project back on schedule. This accounts for the “double horn” pattern in Actual Fraction of a Man-Day on Project in the figure.

Concluding Comments on the Model of the EXAMPLE project.

In this chapter, the experimental setting of EXAMPLE was identified. This enabled experimentation and analysis of the dynamics of software development. The EXAMPLE software project serves as the prototype. EXAMPLE was run and analyzed, and the behavior of several project patterns were presented, explained and found to replicate reported observations in the central literature.

This indicates that EXAMPLE is ready to be used as a laboratory tool to study the dynamic implications of managerial actions, policies and procedures on areas such as scheduling, controlling, QA and staffing.

Summary Chapter 16

Estimation by analogy

The 16th chapter focus on estimation by analogy. In the previous chapters of the book the main focus was quantitative software estimation methods. In this chapter however, the focus will be turned towards the analogy method of estimation that is the most commonly used method to estimate software projects. Estimation by analogy involves as the name depicts, reasoning by analogy with one or more completed projects to relate their actual costs to an estimate of the cost of a similar new project. Hence one builds the estimation for a new project in analogy with similar completed projects.

To apply this method, one need then at least one completed project with similar features. The new project must also have been clearly specified at a functional level in order for it to be compared with similar element.

Oliver argues that the most common method to operationalize and calculate project estimates is to build on previous project experience. This notion is supported by Thayer’s PhD dissertation where 60% of the 60 projects within the aerospace industry applied the analogy method in order to calculate project estimates. This makes it by far the most common method of estimation in Thayer’s study.

Abdel-Hamid argues through the previous chapters of the book that software project estimation affects project behaviour. A project’s estimate creates pressures and perceptions that directly influence the decisions taken by participants through the lifecycle of the project. For example hiring and firing policies are constructed based on perceived project status and productivity. This implies that the use of analogy in estimation results in a feedback loop as depicted in the figure below.

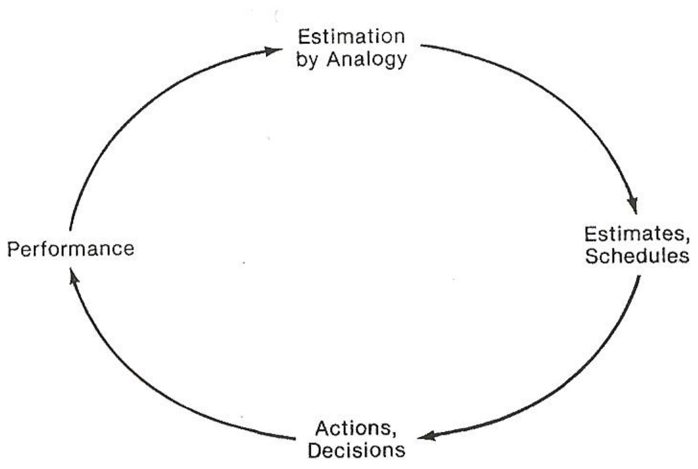


Figure 16.2 Feedback loop resulting from estimation by analogy

As the figure show the analogy method produces estimates that affect decisions and actions of the technical performers and managers which in turn affect work performance. This in turn influence future estimations, and the cycle is complete. In order to identify the effect of estimation by analogy feedback loop, and how they influence projects in a positive or negative way the authors create a hypothetical situation in which a company undertakes a sequence of five identical software projects. These are all identical to the EXAMPLE prototype used in the thesis.

EXAMPLE1 uses the base case conditions. In this scenario the company is lacking previous project experience, and thus underestimates the size of the project by 33%. The company estimates the project to be 48.22 KDSI. The project's man-days are estimated to be 2359, and its development time to be 296 working days. As in the base-case analysis in chapter 13 the analysis of EXAMPLE1 provides the following lesson:

1. The project is really 64KDSI
2. It consumes 3795 man-days
3. It takes 430 days to complete.

<u>EXAMPLE 2</u>

Example 2 is identical to example above, but it comes along later and the project management is in a better position to estimate its true size. We therefore assume that EXAMPLE 2 is estimated perfectly to 64KDSI, requires 3795 man-days and lasts for 430 days. Based on these figures the workforce required is calculated by management to be 9 full-time employees.

When running EXAMPLE 2 under these circumstances, it yields the following result:

1. The project's size is 64KDSI
2. Actual Man-days needed: 3787
3. Actual Duration 454 days

While the project is almost perfectly on target in terms of man-day expenditures it still finishes late. It finishes approximately 6% beyond the estimated schedule. This is a surprising and disturbing result as EXAMPLE 2 overruns despite being "perfectly" schedule estimated.

Analysis of the Experiments

The above mentioned sequence of actions was repeated for EXAMPLE 3, 4 and 5. The observed surprising behaviour persisted through each simulation, were the schedule was overrun in every case. Management started each project with a slightly longer scheduled duration than the previous, yet this yielded the same overrun result.

The interpretation of the conducted simulation allows two interpretations of the experiment's result. First there appears to be inherent factors in the management of a software project that causes it to overrun even at "perfect" schedule estimate. The second is that because of the inherent tendency to overshoot, the use of the analogy method injects a bias in estimation. This bias creates in the long run, longer than necessary schedules.

EXAMPLE 2 overran a schedule that was perfectly estimated within the needs provided by EXAMPLE 1 that was completely identical. Through further experimentation the author's discover the root to the problem. A project runs into scheduling problems when "Man-Days Perceived Still Needed" exceeds the "Man-Days Remaining". For example, if the project's size was underestimated the value of the "Man-Days Perceived Still Needed" could rise as the undiscovered tasks are found. This will not happen in EXAMPLE 2 as it is thought to be perfectly estimated by the experiences from EXAMPLE 1. What may happen is that the value of "Man-Days Remaining" will drop below the "Man-Days Perceived Still Needed". This creates a schedule-problem for EXAMPLE 2. If this happens late in the project, and turnovers are decreasing the work-force level, management will be reluctant to hire new people to the project. The adjustment will therefore be to push the completion date back, and the result is the observed schedule overrun.

The second observation considers the inherent tendency to overshoot. It can be stated as follows: because of the inherent tendency to overshoot, the use of the analogy method in estimating injects a bias in an organization’s scheduling that create in the long term, longer than necessary project schedules.

This phenomenon has been encountered frequently in system dynamics studies of organizational behaviour. It has been termed “The Policy Resistance of Social Systems” This can be illustrated by the caffeine addiction example. A caffeine addict needs a certain dose to generate a given level of awareness. Over time the burden of maintaining alertness shifts from the body’s own physiological processes to the externally supplied caffeine dose. Over time the results is an increase in caffeine doses in order to maintain the same level of alertness.

Richardson and Pugh provide an explanation for why social systems have this tendency of resistance. They argue that the compensating feedback property of real systems as well as system dynamics models tend to be resistant to policies designed to improve behaviour. A parameter change may weaken or strengthen a feedback loop, but the multi-loop nature of a system dynamics model naturally strengthens or weakens other loops to compensate. The result is often little or no overall change in model behaviour. This is exactly what happens in EXAMPLE 2 3 4 5.

EXAMPLE 2 estimates a average staffing level of 8.8 fulltime employees. MD=3795, TDEV=430days. We know that the man-days needed are in fact 3787 and the Total development time is 454. The actual staffing level is 8.3 rather than the reported 8.8 due to the turnover problem. When the analogy method is implied to EXAMPLE3 the actual lower level is passed on to the next project. Extending the project’s schedule from 430 to 454 days weakens the schedule pressure in the system, and the hiring loop simply compensates by decreasing the project’s starting workforce. Such compensating behaviour often og undetected by the participants in the project, and its unlikely that a manager would notice the compensation as the 8.8 number is an estimation from EXAMPLE 2. Therefore the EXAMPLE 3 project is poorly estimated from the beginning, and it will cause an overrun due to the compensating behaviour of the feedback loops. When a software development project overruns its schedule there is an apparent cause: the project was poorly estimated.

<u>Three conclutions</u>

1. A software development project can still overrun despite a “perfect” schedule estimate.
2. Research on the causes of schedule overrun problems must be extended beyond merely estimation accuracy. One such cause is the interaction of the manpower-acquisition policy and turnover rate.
3. Estimating by analogy injects a bias in an organization’s scheduling process that generates longer than necessary project schedules.

Summary Chapter 17

<u>The 90% syndrome</u>

The 90% syndrome is a control problem that is faced by many software project managers. There is ample evidence in the literature to support the notion of such a syndrome, and the 17th chapter uses an EXAMPLE to provide insights into it. Barber describes the 90% syndrome problem as “ …estimates of the fraction of work completed increases as originally planned until a level of about 80-90% is reached. The programmer’s individual estimates then increase only very slowly until the task is actually completed.

In order to simulate and explain the 90% syndrome three EXAMPLE cases were created with three different initial values. This was done in order to verify how the problems arises when certain aspects of the project is underestimated.

The base case was initially underestimated by 33%, giving the following estimations:

1. 42.88 KDSI (lower than the actual 64)
2. 2359 man-days
3. 296 total development days

The second EXAMPLE case gave a perfect estimation of the project’s size but man-day requirements are underestimated by 33%. Such a situation may arise because of an underestimation of a project’s complexity or an overestimation of the team’s productivity.

1. 64 KDSI

2. 2407 man-days
3. 299 total development days

The third and final EXAMPLE is a case where neither size nor man-day requirements are underestimated:

1. 64 KDSI
2. 3593 man-days
3. 358 total development days

Size underestimation

The 90% syndrome, as might be expected, arises when a software project is initially underestimated. Progress generally lacks visibility in the earlier stages and is measured by the expenditure of resources rather than achievement. Status reporting at this point in time is merely an echo of the project’s plan, giving it an illusion of being on track. As the project enters its final stages discrepancies between the percentage of tasks accomplished and the percentage of resources expended are increasingly apparent. Management and workforce become increasingly aware of the actual productivity and the workload remaining. Thus even as the project members proceed towards the final stages of development at a higher work rate they discover that more and more work needs to be done and that their effective progress towards completion is reduced.

Man-Day Underestimation

Initially the author’s did not expect significant differences in the acuteness of the 90% problem between the size-underestimation and man-day underestimation. The EXAMPLE however shows that the 90% syndrome is much more acute when man-day requirements are underestimated. The reason for this is clearly visible with a little reflection. When man-days are underestimated the problem often remains undetected until the final stages when most of the budget man-days are consumed and the project members can perceive how productive the work force has actually been. When size is underestimated it tends to be detected faster, and when the final stages approaches the syndrome is largely visible. This reduces the acuteness of the 90% syndrome in cases were the size has been underestimated. For man-days however the detection rate is significantly slower, and the problem is acute.

Implication of the 90% syndrome

The 90% syndrome arises due to the interaction of two factors, namely underestimation and imprecise measurement of progress. Progress tends to be imprecisely measured because imprecise surrogates are used to measure it. A surrogate is a substitute measure for a phenomenon that cannot be feasibly measured directly. In software development, reports suggest that consumption of resources is a surrogate measure for progress.

There have been attempts to make better tools for directly measuring and monitoring systems that would be able to measure progress. They have unfortunately limited success as they tend to focus on only one aspect of the problem, namely the imprecise measure of progress. The problem of underestimation is not covered by these automated systems and using them may therefore lead to unintended and dysfunctional consequences. The better the measurement tool, the earlier it will detect that progress is not keeping up with the underestimated schedule. When a discrepancy is detected early in the cycle, management usually reacts by adding more people rather than adjusting the schedule. This happens according to DeMarco for political reasons: Once an original estimate is made, it's all too tempting to pass up subsequent opportunities to estimate by simply sticking with your previous numbers. This often happens even when you know the old estimates are substantially off.

The result of sticking with a schedule that is too tight is an increase in project cost due to a bloated work force level. This in turn produces a project that is erroneously compressed in duration and inflated in cost.

The chain effects in going from a problem to immediate consequences, then to second order consequences and newly created problems is a pervasive characteristic of modern social systems. Quite literally in such systems everything depends on everything else and often in ways so complex and roundabout that it is difficult to understand the interrelationships. The result is a situation where logical solutions may prove faulty as their consequences ramify. Since the consequences appear later than the decision itself, it is very hard for members of the team to backtrack and trace the disruptive consequence to the decision that fostered it.

Summary Chapter 18

The economics of quality assurance

Quality Assurance is a series of activities performed in the development of a software product to ensure that the doubts and risks concerning its performance in the target environment. Software quality assurance is approached by to distinct and complementary methodologies, where the first method is to assure that the quality is initially built into the product. This method emphasizes coherent, complete, unambiguous and non-conflicting set of requirements designed early in the project. The second method is to review and test the product continuously as it is designed and coded.

Given the model boundaries by the authors the method that will be focused upon in this chapter concerning the economy of QA, is the second approach. Several specific techniques are available for reviewing and testing the software product as it is designed and coded. These include structured walk-through and technical reviews, inspections, code reading and integration testing.

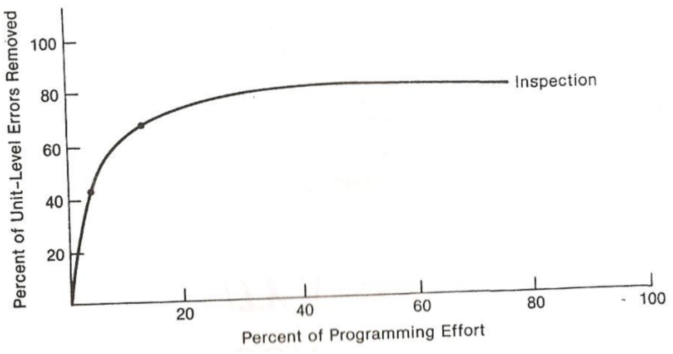
In this chapter the focus is on the economy of QA rather than the technical aspects. The chapter investigates the tradeoff between the benefits and costs of the QA effort in terms of total project costs.

QA tools and techniques add direct cost to the development of software. Man-hours are expended in developing test cases, running them and conduct structured walk-throughs. The cost of QA is a concern for both management and customer. It is also a pressing concern to the software quality manager how cost efficient the QA effort is in the middle of the software production cycle. This pressing matter has not, however, been addressed in the literature. There are no published studies investigating how **cost efficient** QA operations are during the production cycle. There are three possible reasons for this knowledge gap: (1) It is a managerial issue. As in all aspects of software development, managerial issues attract less attention. (2) The QA effort has been largely emphasized as a selling point to managers focusing on the benefits. (3) The high cost of controlled experimentations in software engineering has made it difficult to perform such research. The question is not if QA should be conducted but much QA is justified.

Cost savings

The primary goal of the QA effort is to detect errors and correct them as early as possible, so that a minimum amount of errors slip through from one development phase to the next. Several studies have established the significant savings gained by detecting and correcting errors early in the development stages. Shooman determined that detecting and correcting an error in the design phase requires one-tenth of the effort needed, for the same error to be detected and corrected in the testing phase because of the additional inventory of specifications, code and manuals that would also need correction. The important relationship is therefore the one between the **QA effort expended** and the **percentage of errors detected** during development.

A significant feature of the relationship above is the diminishing returns as the QA expenditure exceed 20-30% of the development effort. It is observed in the literature by for example Shooman that in any sizeable program it is impossible to remove all errors. The QA effort will be effective at detecting errors up to a certain point. From this break-off point the curve will flatten out as the increase in QA effort still cannot detect all errors. This is presented in the figure below by Boehm's QA study.



This suggests that there is a tradeoff between QA savings and cost. The line above flattens out as the increased effort for QA is leveling off at the maximum detection level. However, the allocation of staff for QA increases the workforce exponentially as the increased QA effort requires a larger staff. This in turn leads to more newly hired staff that is less productive than seasoned workers. Therefore it is incredibly important for management to balance the ratio between the QA effort and overall production costs.

In order to exemplify this relationship Tarek Hamid turns again to an EXAMPLE in order to provide the relationship between cost and benefit of QA.

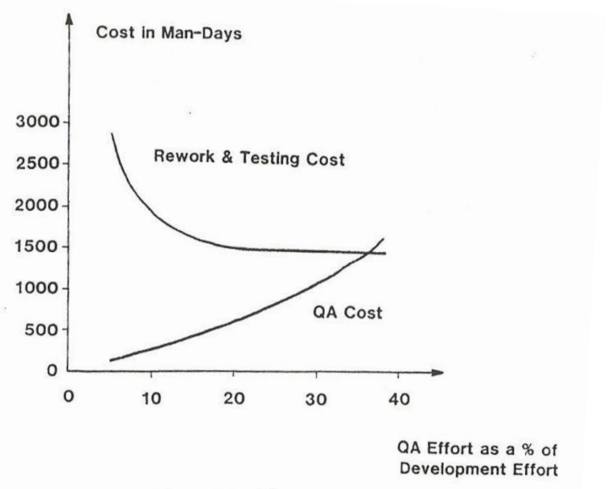


Figure 18.3 QA effort impact on costs

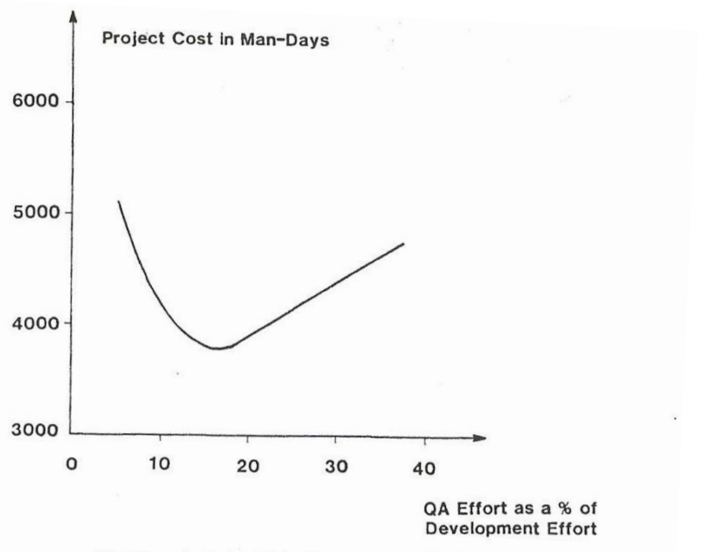


Figure 18.4 Optimal QA effort for project EXAMPLE

In figure 18.3, as the QA costs rise, the benefits decline. In order to find the "optimal" balance for EXAMPLE, the total cost in man-days are plotted against the QA effort defined in terms of percentage of development man-days. The optimal QA effort is 16% as illustrated in figure 18.4

Two important conclusions can be drawn from the second figure. **Firstly, the QA policy has a significant impact on total project cost.** EXAMPLE's total costs range from a low of 3370 man-days to values around 5000 man-days. The high values are 33% higher than the low point. At low levels of QA, expenditures increase due to the large costs of the testing phase as errors have gone undetected and slipped through. At high values of QA the excessive QA expenditures are the reason for increased costs.

Secondly the optimal result of 16% derived from the model is of importance. The significance is not the value itself as there is no generality beyond the EXAMPLE case, however the EXAMPLE show that the process of deriving at 16% strengthens the integrative system dynamics approach. This is the first model that can analyze quantitatively the costs and benefits of QA for software production. The model is therefore generalizable because one can customize models for different software development projects

Generality of Optimality

The optimal value of 16% derived above is not in itself generalizable. In order to illustrate this point two EXAMPLE experiments will be conducted. In these experiments two project variables that can vary between project and organizations will be utilized. Such an investigation will provide two useful outcomes. (1) The experiment will derive results from the form “An increase in X warrants a greater QA-expenditure” which will be generalizable beyond EXAMPLE. Such results could be formulated as useful “rules-of-thumb”. (2) Using “rules-of-thumb” we can adapt and adjust our own results, increasing their generalizability.

The first EXAMPLE is a scenario where the effect on QA of a 40-20-40 effort distribution is examined. 40% is allocated for preliminary and detailed design, 20% for coding, 40% for testing. This distribution is perhaps the most widely touted rule of thumb on distribution of effort among phases in a software development project. In our model a 40-20-40 translates into a 60-40 distribution. 60% of total man-days are allocated design and coding, 40% to testing.

Running EXAMPLE where 40% of the man-days are allocated to testing rather than 20% as in the base-case, yield an optimal QA expenditure of 11% of the development effort. This is a clear decrease from the 16% yielded in the base-case. The fundamental reason for this is that the effort expenditures are a function not only of the actual workload but also of planned expenditures. Allocating more to testing expands the testing effort even though the workload may not. Since the testing effort will expand as a result of management’s increased allocation to testing, it makes sense to reap the greatest return from the increased testing investment. This is achieved by cutting the QA effort.

In the second EXAMPLE the author made a clear distinction between two sets of factors that can affect how productive people will be on a software project. The first set, included factors that affect productivity dynamically throughout the development of a single project. These factors include work experience, learning, motivation, and communication. The second set included environmental factors that tend to vary during the life-cycle of a project. Availability of software tools, computer-hardware characteristics, programming language and product complexity. These factors do not play a dynamic role during the life of a single project, and is therefore represented by a single parameter. This parameter is named "Nominal Potential Productivity". Here the effect on optimal QA by changes in the nominal potential productivity rate is investigated.

The results of the experiment show that an increase in development productivity warrants an increase in QA expenditures relative to development expenditures. Higher development productivities mean that each man-day expended yield more software. The increased output requires more QA effort. Note that the increased output itself is not triggering an increase in the QA effort. The required increase in the QA effort must explicitly be planned for. This was addressed in chapter 8, where the Parkinsonian nature of the QA effort was identified. QA is planned and executed accordingly. No matter how many tasks needed to be processed within a QA review, each QA almost always processes all tasks. This follows Parkinson's law, and even when QA is relaxed or suspended no backlog develop. When walkthroughs are suspended the requirement for a walkthrough is also suspended not postponed.

<u>The Search for the Optimal QA effort</u>

There are two conclusions drawn from these experiments. The first is that QA holds a significant impact on the total project cost. The second is that although the optimal QA percentage is dependent on the environment it is possible to represent the characteristics of the environment in our model to ascertain the appropriate QA allocation for a specific project.

Summary Chapter 19

Brook’s law

Through the book, Abdel-Hamid’s integrated dynamic model has increased the understanding of software production management. It has established the viability of the system dynamics methodology as a research vehicle. The 19th chapter focuses on Brook’s law, and through model enhancements attempt to reproduce it in the dynamic model.

Dr. Fred Brook first publicized his law in the book “The Mythical Man-Month: Essays on Software Engineering.” The book embodies many insights into the management of large software projects gained through his own experience in managing the development of IBM’s OS/360. Brook’s Law is stated as follows: Adding manpower to a late software project makes it later. The lack of interchangeability between men and months was recognized as being caused by training and intercommunication overheads.

Brook’s law has been endorsed by the software literature, and it has often been used in both large scale and small scale projects. Brook himself specifically tailored the Law to what he identifies as “jumbo” projects, however, scholars such as Pressman extend it to small 6-10 man-year projects. Interestingly enough, the widespread endorsement of Brook’s law has taken place even though it has never been formally verified. In order to verify if Brook’s law can be applied to medium ranged software projects, the authors turn once again to their EXAMPLE prototype base-case.

Model Enhancements

In order to highlight Brook’s law, several enhancements of the model are necessary. Many studies have identified how communication and training overheads have a negative impact on productivity. Figure 19.1 show the original model where manpower is divided into newly hired workforce and experienced workforce.

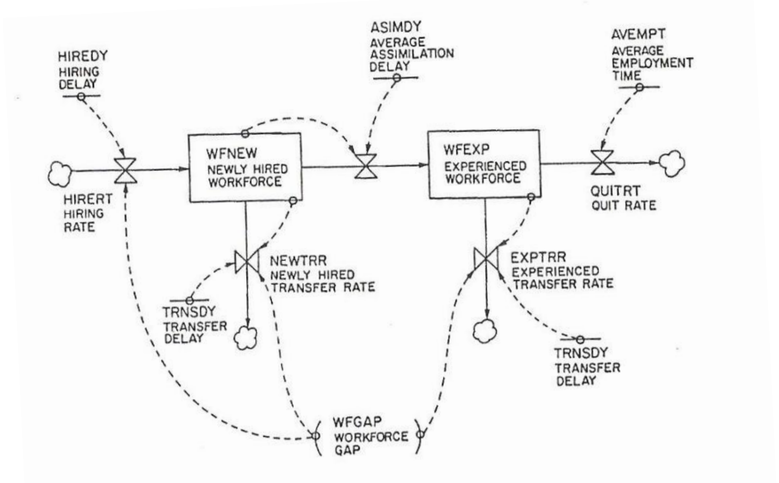


Figure 19.1 Original model of manpower assimilation (hiring and training)

However, this is a simplification of the assimilation of the workforce. In reality, while the average productivity of new hires may be half of an experienced worker at the time of hiring, the productivity will increase through learning. The initial low productivity of new hires is important to represent in a study of Brook's Law since immediately after new people are added, the net impact on productivity may be negative. In a production environment where the progress of development is perfectly visible, information flow is instantaneous, and management's reactions are immediate. Management's policy if a project falls behind is to hire more employees. Such a negative impact on the production rate may cause a vicious cycle.

The first enhancement to the model is a multiple level system of manpower assimilation. Instead of one level of "WFNEW", there have been constructed three levels namely WFNEW1, 2 and 3. Separating hired personnel into three levels allows the capturing of productivity differential that exists between the time of hiring to the end of training. The table below captures the calculated values of productivity for each level of the newly hired workforce.

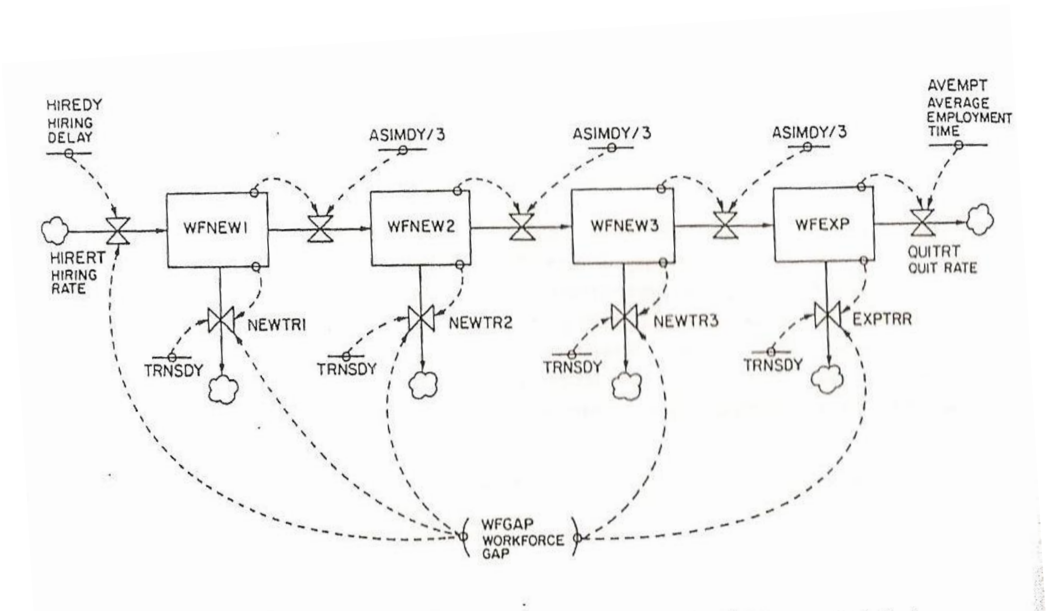
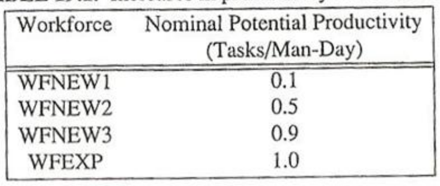


Figure 19.2 Enhanced model of manpower assimilation (hiring and training)

Reforumlation

The new average nominal potential productivity for newly hired workforce is still 0.5, half of a seasoned worker. The values of potential productivity are given in the table below.

TABLE 19.1. Increases in productivity due to training



Workforce	Nominal Potential Productivity (Tasks/Man-Day)
WFNEW1	0.1
WFNEW2	0.5
WFNEW3	0.9
WFEXP	1.0

With the new modifications set to the model, several reformulations of Average Nominal Potential Productivity need to be set. Four different values need to be calculated rather than just two in the original model. The original formulation was:

$$ANPPRD = \frac{WFEXP \times NPWPEX}{TOTWF} + \frac{WFNEW \times NPWPNE}{TOTWF}$$

where

$WFEXP$ = *Experienced Workforce*
$WFNEW$ = *Newly Hired Workforce*
$TOTWF$ = *Total Workforce*
$NPWPEX$ = *Nominal Potential Productivity of Experienced Employes*
= *1 Task/Man-Day*
$NPWPNE$ = *Nominal Potential Producitivty of New Emplyess*
= *0.5 Task/Man-Day*

The new formulation of Average Nominal Potential Productivity with the added structure is as follows:

$$ANPPRD = \frac{WFEXP \times NPWPEX}{TOTWF} + \frac{WFNEW1 \times NPWPN1}{TOTWF} + \frac{WFNEW2 \times NPWPN2}{TOTWF} + \frac{WFNEW3 \times NPWPN3}{TOTWF}$$

where,

$NPWPN1$ = *Nominal Potential Productivity of WFNEW1*
= *0.1 Tasks/Man-Day*
$NPWPN2$ = *Nominal Potential Productivity of WFNEW2*
= *0.5 Tasks/Man-Day*
$NPWPN3$ = *Nominal Potential Productivity of WFNEW3*
= *0.9 Tasks/Man-Day*

Reformulation of Nominal Testing Manpower Needed per Error

In addition to the reformulation above, another reformulation needs to be made. In the current model the effort needed to detect and correct errors in the testing phase is formulated as the product of the “Error Density” and the “Nominal Testing Manpower Needed per Error”. The value of the latter was set at 0.15 Man-days/Error.

The formulation for the “Nominal Testing Manpower Needed per Error” does not seem to account for the experience mix of the workforce. That is, TMPNPE = 0.15 whether the project’s workforce is all experienced, or inexperienced. This is not so because TMPNPE is assumed to not be affected by the experience mix of the workforce, but rather an assumption that as the project approaches the testing phase management will be reluctant to hire new personnel. Thus at the testing-phase most personnel will be experienced. The simplified TMPNPE formulation is:

$$TMPNPW = \frac{0.15}{AMPPRD}$$

where,

$$ANPPRD = Average\ Nominal\ Potential\ Productivity$$
$$= 1\ Task/Man\text{-}Day$$

(where we assume that at the testing phase all staff are experienced).

To investigate Brooks' Law, we need to allow for the hiring of new staff even toward the final stages of the project. In such a case, the above assumption does not hold. We therefore change the formulation of TMPNPE from:

$$TMPNPE = 0.15\ Man\text{-}Day/Error$$

to:

$$TMPNPE = \frac{0.15}{ANPPRD}\ Man-Day/Error.$$

If management refrains from adding new staff in the testing phase,

$$TMPNPE = \frac{0.15}{1} = 0.15\ Man-Day/Error$$

as originally formulated. But if management hires new staff in the testing phase, then ANPPRD will be less than 1 Task/Man-Day. The result will be a "Nominal Testing Manpower Needed per Error" greater than 0.15 Man-Day/Error. This creates a loss in testing productivity due to the lower average experience of the workforce.

Modeling more aggressive manpower acquisition policies

Brook's Law entails a scenario where the management's reaction to a project fallen behind schedule, is continuous hiring of new personnel in order to get the job done within the deadline. In EXAMPLE management's hiring-policy is expressed in the model by "Willingness to Change workforce". By adjusting this variable, one can examine the impact of a more aggressive hiring-policy on the project's overall cost and duration. It also allows us to build two separate scenarios, where A holds a more aggressive policy than B. In EXAMPLE B management is more reluctant to hire new staff late in the project, and should therefore not experience to the same degree Brook's Law.

In the base case presented earlier, "Willingness to Change Workforce" is the sum of "Hiring Delay" and "Assimilation Delay". In the early stages when "Time Remaining" is generally higher than the sum of "Hiring Delay" and "Assimilation Delay", management is more open to adjust the workforce in order to meet the scheduled completion date. The delays are set for 40 and 80 working days respectively.

When "Time Remaining" drops below 180 days left management in the base case starts to be reluctant to adjust the workforce. When "Time Remaining" drops below 48 days no more additions to the workforce will be carried out.

In EXAMPLE the "Willingness to Change Workforce" is set to 30 days, which is a much more aggressive strategy. This leads to a scenario were the willingness to hire new employees carries on much later into the project. Management will continue to be inclined to adjust the workforce until 45 days remain, at this point they will start to be reluctant to add new members to the workforce. Management will make no additions to the workforce when only 9 days remain before the scheduled completion date. This scenario should identify Brook's Law clearly.

<u>Results from the More Aggressive Manpower Acquisition Policies</u>

The results from adjusting the value of the time parameter is shown in the graph below

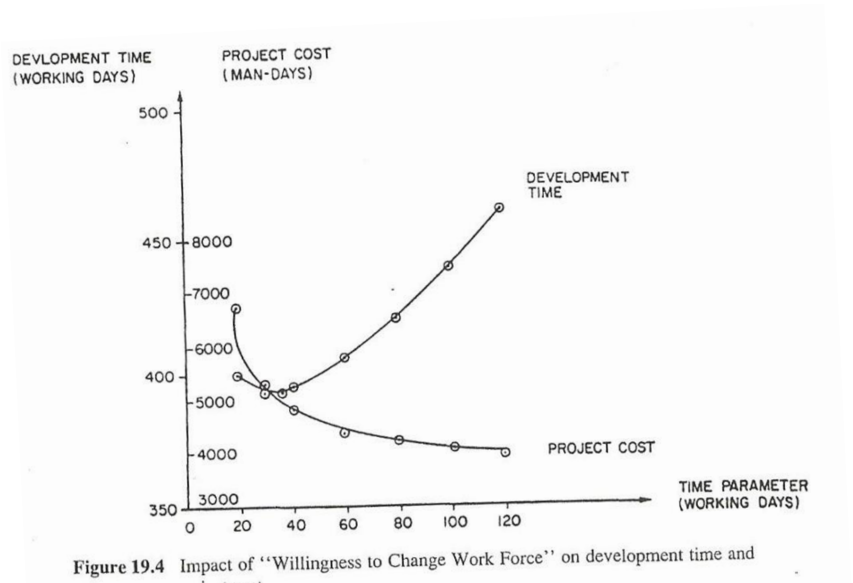


Figure 19.4 Impact of "Willingness to Change Work Force" on development time and project cost

As we can see it does not totally support Brook's Law. Adding more people to a late project causes it indeed to become more costly, but not always further delayed. The increase in cost is caused by the increased training and communication overheads. This in turns affects average productivity negatively which in turns increase the project's man-day requirements.

For the project's schedule to suffer, the drop in productivity must be large enough to render an additional person's net cumulative contribution to the project to be negative. A new hiree's immediate net contribution might be negative due to the need for training. Experienced Man-days are also diverted from the project as experienced personnel train new hirees. However, as the new hirees learn, their cumulative net contribution rises and provides positive addition to productivity. Brook's Law only comes into effect when the cumulative impact is negative from newly hired personnel and the project is completed at a later time.

Impact on Aggressive Manpower Acquisition Policies on Productivity

Figures 19.5 and 19.6 plot the workforce levels and Average Nominal Potential Productivity for model runs with "Time Parameter" values of 30 and 20 working days.

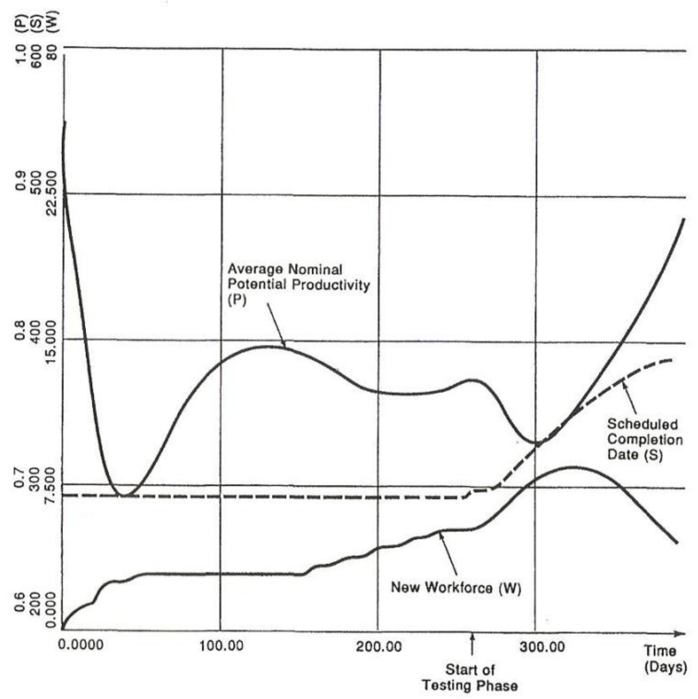


Figure 19.5 Aggressive manpower acquisition policy (TMPR = 30)

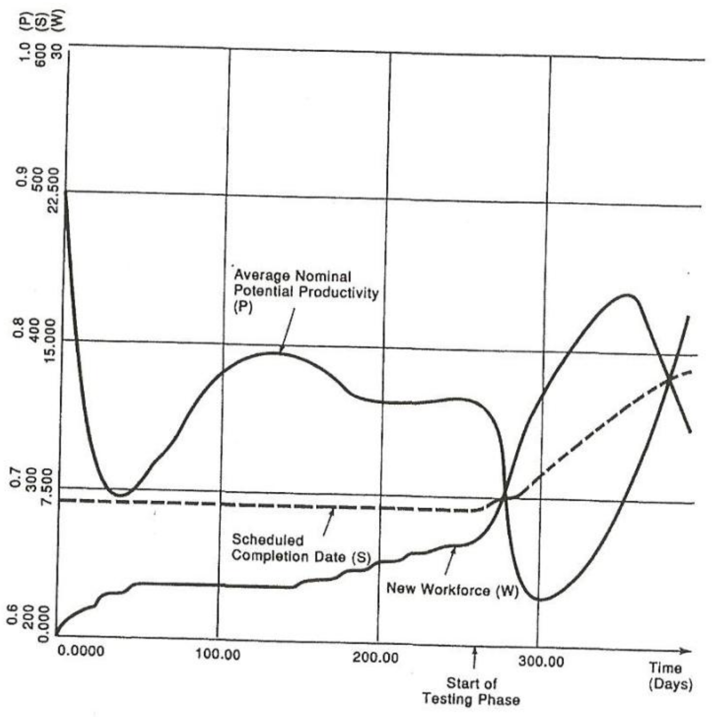


Figure 19.6 Very aggressive manpower acquisition policy (TMPR = 20)

The time parameters of 30 and 20 working days are the values for which Brook's Law applies. The results above shed some light on why the extremely aggressive manpower acquisition policy of 20 working days delays the completion date opposed to the relatively milder 30 working day "Time Parameter"

In both cases we see that significantly increasing the number of new hires in the testing phase leads to an immediate and significant drop in Nominal Average Potential Productivity. However, as we see in figure 19.6 that more new personnel continue to be hired until later in the life cycle. This is the result of a lower "Time Parameter" value.

By the time EXAMPLE reaches the testing phase, it is evident to management that the project is lagging behind schedule. Their option is to hire more people, extend the schedule or a combination of both. Under the more aggressive manpower acquisition policies, management is less reluctant to add manpower to minimize schedule disruptions. We see this in figure 19.6 as management chooses to keep the scheduled completion date slightly longer than in figure 19.5. The choice of keeping a tight schedule leads to only one possible policy, namely hire more people. A larger workforce however means more training and communication overheads. This leads to a diminished productivity rate that in turn leads to a need for a larger workforce. Thus EXAMPLE B with a Time Parameter of 20, more new people is hired later in the life cycle. This explains why the dip in Average Nominal Potential Productivity observed in figure 19.6 above is so severe. The dip in productivity is severe and delayed enough to prevent the new hires from ever reaching a high level of productivity. Therefore in this scenario the net cumulative project contribution is never converted from negative to positive. This causes the project to be delayed and finish later as described by Brook's Law.

Contribution

This chapter has provided two essential contributions. First it has given a feel for why and how to enhance the model's formulation in order to investigate new issues of software project management that lie beyond the purpose and scope of the original model. Secondly the experimental results provide insights into Brook's Law in medium ranged software projects. Adding more people to a late project will make it more costly, but not necessarily cause it to finish later. For the project's schedule to suffer, the drop in productivity in the project must be delayed and severe enough to render an additional worker's net cumulative contribution negative. This happens as demonstrated above only in cases where management holds an aggressive hiring policy. chapter concerning the economy of QA, is the second approach. Several specific techniques are available for reviewing and testing the software product as it is designed and coded. These include structured walk-through and technical reviews, inspections, code reading and integration testing.

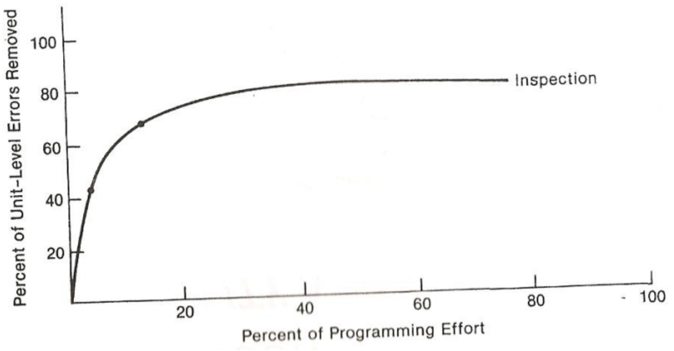
In this chapter the focus is on the economy of QA rather than the technical aspects. The chapter investigates the tradeoff between the benefits and costs of the QA effort in terms of total project costs.

QA tools and techniques add direct cost to the development of software. Man-hours are expended in developing test cases, running them and conduct structured walk-throughs. The cost of QA is a concern for both management and customer. It is also a pressing concern to the software quality manager how cost efficient the QA effort is in the middle of the software production cycle. This pressing matter has not, however, been addressed in the literature. There are no published studies investigating how **cost efficient** QA operations are during the production cycle. There are three possible reasons for this knowledge gap: (1) It is a managerial issue. As in all aspects of software development, managerial issues attract less attention. (2) The QA effort has been largely emphasized as a selling point to managers focusing on the benefits. (3) The high cost of controlled experimentations in software engineering has made it difficult to perform such research. The question is not if QA should be conducted but much QA is justified.

Cost savings

The primary goal of the QA effort is to detect errors and correct them as early as possible, so that a minimum amount of errors slip through from one development phase to the next. Several studies have established the significant savings gained by detecting and correcting errors early in the development stages. Shooman determined that detecting and correcting an error in the design phase requires one-tenth of the effort needed, for the same error to be detected and corrected in the testing phase because of the additional inventory of specifications, code and manuals that would also need correction. The important relationship is therefore the one between the **QA effort expended** and the **percentage of errors detected** during development.

A significant feature of the relationship above is the diminishing returns as the QA expenditure exceed 20-30% of the development effort. It is observed in the literature by for example Shooman that in any sizeable program it is impossible to remove all errors. The QA effort will be effective at detecting errors up to a certain point. From this break-off point the curve will flatten out as the increase in QA effort still cannot detect all errors. This is presented in the figure below by Boehm's QA study.



This suggests that there is a tradeoff between QA savings and cost. The line above flattens out as the increased effort for QA is leveling off at the maximum detection level. However, the allocation of staff for QA increases the workforce exponentially as the increased QA effort requires a larger staff. This in turn leads to more newly hired staff that is less productive than seasoned workers. Therefore it is incredibly important for management to balance the ratio between the QA effort and overall production costs.

In order to exemplify this relationship Tarek Hamid turns again to an EXAMPLE in order to provide the relationship between cost and benefit of QA.

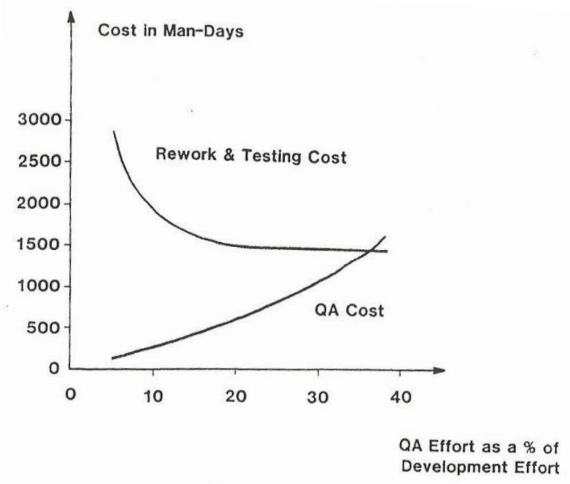


Figure 18.3 QA effort impact on costs

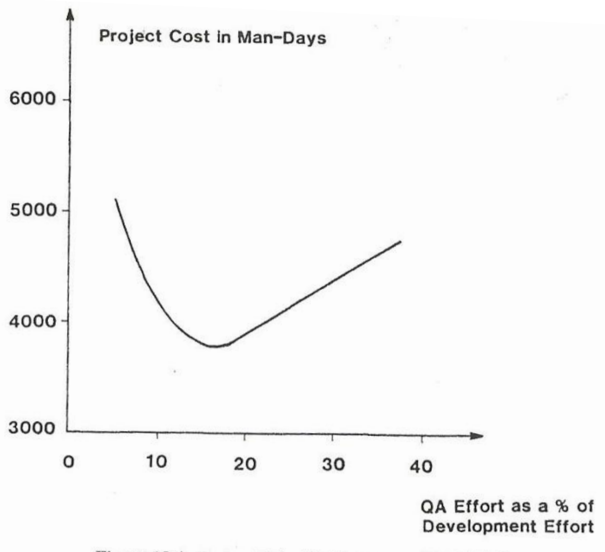


Figure 18.4 Optimal QA effort for project EXAMPLE

In figure 18.3, as the QA costs rise, the benefits decline. In order to find the "optimal" balance for EXAMPLE, the total cost in man-days are plotted against the QA effort defined in terms of percentage of development man-days. The optimal QA effort is 16% as illustrated in figure 18.4

Two important conclusions can be drawn from the second figure. **Firstly, the QA policy has a significant impact on total project cost.** EXAMPLE's total costs range from a low of 3370 man-days to values around 5000 man-days. The high values are 33% higher than the low point. At low levels of QA, expenditures increase due to the large costs of the testing phase as errors have gone undetected and slipped through. At high values of QA the excessive QA expenditures are the reason for increased costs.

Secondly the optimal result of 16% derived from the model is of importance. The significance is not the value itself as there is no generality beyond the EXAMPLE case, however the EXAMPLE show that the process of deriving at 16% strengthens the integrative system dynamics approach. This is the first model that can analyze quantitatively the costs and benefits of QA for software production. The model is therefore generalizable because one can customize models for different software development projects

Generality of Optimality

The optimal value of 16% derived above is not in itself generalizable. In order to illustrate this point two EXAMPLE experiments will be conducted. In these experiments two project variables that can vary between project and organizations will be utilized. Such an investigation will provide two useful outcomes. (1) The experiment will derive results from the form "An increase in X warrants a greater QA-expenditure" which will be generalizable beyond EXAMPLE. Such results could be formulated as useful "rules-of-thumb". (2) Using "rules-of-thumb" we can adapt and adjust our own results, increasing their generalizability.

The first EXAMPLE is a scenario where the effect on QA of a 40-20-40 effort distribution is examined. 40% is allocated for preliminary and detailed design, 20% for coding, 40% for testing. This distribution is perhaps the most widely touted rule of thumb on distribution of effort among phases in a software development project. In our model a 40-20-40 translates into a 60-40 distribution. 60% of total man-days are allocated design and coding, 40% to testing.

Running EXAMPLE where 40% of the man-days are allocated to testing rather than 20% as in the base-case, yield an optimal QA expenditure of 11% of the development effort. This is a clear decrease from the 16% yielded in the base-case. The fundamental reason for this is that the effort expenditures are a function not only of the actual workload but also of planned expenditures. Allocating more to testing expands the testing effort even though the workload may not. Since the testing effort will expand as a result of management’s increased allocation to testing, it makes sense to reap the greatest return from the increased testing investment. This is achieved by cutting the QA effort.

In the second EXAMPLE the author made a clear distinction between two sets of factors that can affect how productive people will be on a software project. The first set, included factors that affect productivity dynamically throughout the development of a single project. These factors include work experience, learning, motivation, and communication. The second set included environmental factors that tend to vary during the life-cycle of a project. Availability of software tools, computer-hardware characteristics, programming language and product complexity. These factors do not play a dynamic role during the life of a single project, and is therefore represented by a single parameter. This parameter is named “Nominal Potential Productivity”. Here the effect on optimal QA by changes in the nominal potential productivity rate is investigated.

The results of the experiment show that an increase in development productivity warrants an increase in QA expenditures relative to development expenditures. Higher development productivities mean that each man-day expended yield more software. The increased output requires more QA effort. Note that the increased output itself is not triggering an increase in the QA effort. The required increase in the QA effort must explicitly be planned for. This was addressed in chapter 8, where the Parkinsonian nature of the QA effort was identified. QA is planned and executed accordingly. No matter how many tasks needed to be processed within a QA review, each QA almost always processes all tasks. This follows Parkinson’s Law, and even when QA is relaxed or suspended no backlog develops. When walkthroughs are suspended the requirement for a walkthrough is also suspended not postponed.

The Search for the Optimal QA effort

There are two conclusions drawn from these experiments. The first is that QA holds a significant impact on the total project cost. The second is that although the optimal QA percentage is dependent on the environment it is possible to represent the characteristics of the environment in our model to ascertain the appropriate QA allocation for a specific project.

Summary Chapter 20

Conclusions and Future Directions

The objective of the book was to enhance our understanding and gain insight into the general process of software development, and how it is managed. Abdel-Hamid built an integrative system dynamics model of a software development project, which in turn was subjected to a case study in order to verify its validity and tests its parameters. The model was then used to study and predict the dynamic implications of several managerial policies and procedures.

The model integrated the knowledge of micro components of software development project management into an integrated and continuous view of the software development process as a whole. Before one can claim that complete understanding has been achieved of a system, it is important to show that knowledge of each component can be put together into a total system. An organization can be synthesized which allows for the interactions of all the relevant variables and structural components. An integrated approach helps us achieve an overall understanding.

An integrated approach helps us achieve not only the overall understanding, but also allows for experimentation. Problem diagnoses and solution evaluation has also been possible thanks to the model. An integrated approach both prompts and facilitates the search for multiple and potentially diffused set of factors interacting to cause software development problems. For example schedule overshoot problems are not only a result of underestimation, but also caused by management's hiring/firing policies.

The integrated model identifies feedback mechanisms and uses them to structure and clarify relationships in software project management. While the significance and applicability of the feedback system concept has been sustained in many studies outside software engineering, it remains largely foreign to the software engineering management community. This model may remedy this deficit.

The mathematical formulation of system dynamics models requires that the structural relationships between variables are explicitly and precisely defined. The high degree of explication required in the model helped to identify knowledge gaps in the literature. The twenty seven interviews with software managers from five different organizations allowed the knowledge gaps to be filled. The model thus incorporates new findings about management of software project management, for example on manpower acquisition policies under different schedule considerations.

The Case Study

After the model had been constructed, a case study at a 6th organization was conducted in order to test the accuracy and viability of the model on an empirical case. The DE-A project at the SGS was used to test and run simulations on an already finished project. The project was chosen do to its size, recent completion and its applicability as a typical project.

To simulate the DE-A project the model was first parameterized. 14 model parameters were set, for example “Hiring Delay and “Turn-Over rate”. The parameter values were obtained from interviews at NASA and project documentation.

The model was highly accurate in reproducing the actual development history of the DE-A project. It reproduced the dynamic behavior of the project’s completion-date estimates, man-day estimates, cost in man-days and workforce loading.

Experimentation

After establishing the models viability in a case-study, it was used as a experimentation vehicle to study several managerial policies and procedures. In total three areas were studied.

1. Software cost and schedule estimation

Three experiments were conducted concerning software costs and schedule estimation. The first experiment examined the impact schedules have on project performance. The experiment showed that different schedules create different projects. An important implication of this experiment is the observation that an estimation model for software development cannot be judged solely on its accuracy to estimate historical projects. It is important to consider not only how accurate an estimation model is, but also how costly the project it “creates” is.

The second experiment concerned the portability of quantitative software estimation tools. Literature on this field suggests that previous models have been poor in portability. Models tailored for one specific type of project or organization fail to be of any use outside this environment. The integrated model here however, demonstrates how it is portable from scenario to scenario. Four different aspects of a managerial environment were identified by the author, namely manpower acquisition, manpower allocation, effort distribution and quality assurance. Because these areas and variables are clear to a project manager in the beginning of a project, it is feasible to incorporate them into future cost estimation models. This will improve both accuracy and portability of such models in the future.

The third and final experiment considered the analogy method of software estimation. The experiment yielded two important observations. The first observation is the inherent factors in the management of a software project that cause it to overrun. This was a result of interaction between manpower acquisition policies and personnel turnover. The experiment demonstrated that even when estimated "perfectly" the internal interactions caused overruns.

The second important observation is the fact that the inherent tendency to overshoot, the use of the analogy method of estimations injects a bias in the scheduling process that creates longer than necessary project schedules.

2. The economics of QA

Two sets of experiments were conducted on QA. The objective was to investigate how much QA is justified in a software project. To do so, the author first examined the relationship between the QA expended and the percentage of errors detected. A significant feature of this relationship is the diminishing returns of QA as QA costs extend beyond 20-30% of the development effort. The derived optimal QA expenditure level is the level that minimizes project costs. For the EXAMPLE prototype case this level was 16% of the development effort.

The results from the first set of QA experiments have demonstrated that QA policies have significant impact on total project cost. QA expenditures significantly higher or lower than the derived optimal causes clear increases in the total project cost. Too little QA and the testing phase become more costly. Too much QA and the costs of QA alone consume high percentages of the project's cost.

The objective of the second set of QA experiments was to examine the sensitivity between the distribution of effort between the development and testing phases, and software development productivity. The yielded results constitute a rule-of-thumb that organizations can use to adapt published results or results from other organizations to their own environment.

3. Staffing

The staffing experiment tested the applicability of Brook's Law on medium sized application projects. The Law has been endorsed by the literature, and applied to small and large-scale projects alike. The original Law was formulated for "jumbo" sized projects.

The experiments demonstrate that Brook's Law cannot be applied for the medium ranged software project. Adding more people to a late project will most certainly increase its cost every time, but not cause it to finish later. Hence Brook's law does not universally apply to all software environments. This does not lead to a disqualification of Brook's Law, rather the notion that it is universally applicable to all software project envirnoments.

Future Directions

There are several areas that need to be further researched on the integrated system dynamics model. As shown in chapter 19, in order to investigate the applicability of Brook's Law the structure of workforce assimilation had to be improved. In order to investigate other areas of software development, such modifications will be necessary to broaden the models depth and scope. There are four specific areas suggested by the author himself.

1. *Requirements definition/analysis phase*. This phase is per now outside the boundaries of the system dynamics model. This phase is important to incorporate into the life-cycle of a project, as several studies argue that there are disruptive effects of changes in system requirements on software production and cost.
2. *Multiple projects*. The current model do not allow for simulation of multiple projects developed in parallel with each other. In such a software development scenario the competition for company resources will be a significant dimension. Man power allocation would for example be split between the projects as they both develop.

3. *Large scale environments.* This model was created for medium-ranged software projects. In large scale projects such as DOD software projects that constitutes over a million lines of code, several enhancements of the model will be necessary. For example must the development phase be separated into smaller phases with a set of milestones separating each phase. QA would also need to be enhanced as this effort in large systems is often conducted by independent organizations.

4. *Quality*. Another important extension is the quality of the produced software product. Measures of software quality have to be determined, and several model enhancements must be constructed here. For example would it be interesting to classify errors into different subcategories depending on level of seriousness. A more challenging enhancement would be to capture the effect of motivational factors on overall product quality.

<u>New Modeling Applications</u>

It is also possible to move the focus on software development projects per se, and extend the modeling effort into a broader set of issues pertaining to the software development organization. Rather than tracing life-cycles in software development, one may focus on different department's operations during a project development cycle. There are some questions suggested by the author.

1. *The efficacy of different organizational struggles*
2. *Personnel turnovers, its costs and benefits*
3. *The Impact of management approaches such as Management By Objectives (MBO)*
4. *The organizational/environmental determinants of productivity*

FINALE

A journey of a thousand miles starts with a single step.

References

Abdel-Hamid, T.K., 1991: Software Project Dynamics: An Integrated Approach, Prentice-Hall, Upper Saddle River, USA.

Baber, R.L., 1982: Software Reflected: The Socially Responsible Programming of Computers, Elsevier Science, New York, USA.

Barlas, Y., 1996: Formal Aspects of Model Validity and Validation in System Dynamics, *System Dynamics Review*, 12(3): 183-210.

Bayer, S. & Gann, D., 2006: Balancing Work: Bidding Strategies and Workload Dynamics in a Project-Based Professional Service Organisation, *System Dynamics Review*, 22(3): 185-211.

Black, L.J. & Repenning, N.P., 2001: Why Firefighting Is Never Enough: Preserving High-Quality Product Development, *System Dynamics Review*, 17(1): 33-62.

Boehm, B., 1981: Software Engineering Economics, Prentice-Hall, Englewood Cliffs, USA.

Brooks, Jr., F.P., 1982: The Mythical Man-Month, Addison-Wesley, Boston, USA.

Charette, R.N., 2010: Why Software Fails, *IEEE Spectrum*, http://spectrum. ieee.org/ computing/software/why-software-fails,08.05.2012.

Cougar, J.D. & Zawacki, R.A., 1980: Motivating and Managing Computer Personnel, Wiley, New York, USA.

Crosby, P.B., 1979: Quality is Free, McGraw-Hill, New York, USA.

Davis, S.B., 1974: Quality Management, Prentice-Hall, Englewood Cliffs, USA.

DeMarco, T. & Lister, T., 1999: Productive Projects and Teams, Dorset House, New York, USA.

Deming, W.E., 1982: Out of the Crisis, MIT Press, Boston, USA.

Feigenbaum, A.V., 1991: Total Quality Control, McGraw-Hill, New York, USA.

Ford, D. & Sterman, J., 2003: Overcoming the 90% Syndrome: Iteration Management in Concurrent Development Projects, *Concurrent Engineering Research and Applications*, 111(3): 177-186.

Forrester, J. W. & Senge, P. M., 1980: Tests for Building Confidence in System Dynamics Models, in Legasto, jr., A.A, Forrester, J.W. & Lyneis, J.M. (Eds.), System Dynamics: TIMS Studies in the Management Science, 14, North-Holland, New York, USA, 209-228.

Frank, W.L., 1983: Critical Issues in Software: A Guide to Software Economics, Strategy, and Profitability, Wiley, New York, USA.

Gehring, P.F., jr. & Pooch, U. W., 1977: Lessons Learned From Modeling the Dynamics of Software Development, *Data Management*, Feb.: 14-38.

Ghezzi,C., Jazayeri, M. & Mandrioli, D., 2003: Fundamentals of Software Engineering, Prentice-Hall, Englewood Cliffs, USA.

Glass, R.L., 1979: Software Reliability Guidebook, Prentice-Hall, Upper Saddle River, USA.

Hjertø, G., 2003: In Hjertø, G. & Berg, T., 2003: Kvalitet- og programvareutvikling, Gyldendal Norsk Forlag, Trondheim, Norge.

Homer J.B., Sterman, J.D., Greenwood, B. & Perkola, M., 1993: Delivery Time Reduction in Pulp and Paper Mill Construction Projects: A Dynamic Analysis of Alternatives, Proceedings of the 1993 International System Dynamics Conference.

Imai, M., 1986: Kaizen: The Key to Japan’s Competitive Success, McGraw-Hill, New York, USA (English version).

Jogeklar, N. & Ford, D., 2003: Product Development Resource Allocation with Foresight, *European Journal of Operational Research*, 160(1): 72-87.

Jones, C., 2007: Estimating Software Costs, McGraw-Hill, New York, USA.

Jones, C. & Bonsignour, O., 2012: The Economics of Software Quality, Addison Wesley, USA.

Juran, J.J., 1988: Planning for Quality, The Free Press, New York, USA.

Kahen, G., Lehman, M.M., Ramil, J.F. & Wernick, P., 2001: System Dynamics Modeling of Software Evolution Processes for Policy Investigation: Approach and Example, *The Journal of Systems and Software*, 59: 271-281.

Lee, Z.W., Ford, D.N. & Joglekar, N., 2007: Effects of Resource Allocation Policies for Reducing Project Durations: A Systems Modelling Approach, *Systems Research and Behavioral Science*, 24:551-566.

Lehman, M.M. & Ramil, J.F., 1999: The Impact of Feedback in the Global Software Process, *The Journal of Systems and Software*, 46: 123-134.

Love, P., Edwards, D., Goh, Y. & Han, S., 2011: Design Error Reduction: Toward the Effective Utilization of Building Information Modeling, *Research in Engineering Design*, 22: 173-187.

Lyneis, J.M. & Ford, D.N., 2007: System Dynamics Applied to Project Management: A Survey, Assessment, and Directions for Future Research, *System Dynamics Review*, 23(2/3): 157-189.

McFarlan, F.W., 1974: Effective EDP Project Management, West Publishing, St. Paul, USA.

McKeen, J.D., 1981: The Nature of Inter-Activity Relationships Within the Systems Development Cycle, Queen's University, Ontario, Canada.

Norden, P.V., 1963: Useful Tools for Project Management, Wiley, New York, USA.

Paulk, M.C., 1993: Comparing ISO 9001 and the Capability Maturity Model for Software, *Software Quality Journal*, 2(4): 245-256.

Peters, T. & Waterman, R.H. jr., 1982: In Search of Excellence: Lessons from America's Best-Run Companies, Wiley, New York, USA.

Petersson, H. & Wohlin, C., 1999: An Empirical Study of Experience-Based Software Defect Constant Estimation Methods, *Proceedings International Symposium on Software Reliability Engineering,* Florida, USA, 126-135.

Putnam, L., 1971: A General Empirical Solution to the Macro Software Sizing and Estimating Problem, *IEEE Trans. on Software Engineering* 26-30.

Reichelt, K.S. & Lyneis, J.M., 1999: The Dynamics of Project Performance: Benchmarking the Drivers of Cost and Schedule Overrun, *European Management Journal* 17(2): 135-150.

Repenning, N.P. & Sterman, J.D., 2001: Nobody Ever Gets Credit for Fixing Problems That Never Happened: Creating and Sustaining Process Improvement, *California Management Review*, 43(4): 64-88.

Riehl, R.E., 1977: An Examination of Management Practices in the Development of Business Information Systems, PhD-Dissertation, George Washington University, USA.

Roberts, E.B., 1962: The Dynamics of Research and Development, Harper & Row, New York, USA.

Rus, I., Collofello, J. & Lakey, P., 1999: Software Process Simulation for Reliability Management, *The Journal of Systems and Software*, 46: 173-182.

Schlender, B.R., 1989: How to Break the Software Logjam, *Fortune*, 100-112.

Scott, H. & Wohlin, C., 2008: Capture-Recapture in Software Unit Testing – A Case Study, *Proceedings of Symposium on Empirical Software Engineering and Measurement*, Kaiserslautern, Germany, 32-40.

Shingo, S., 1959: Zero Quality Control: Source Inspection and The Poka-Yoke System, Productivity, Inc., Portland, USA (English version 1986).

Shull, F., Basili, V., Boehm, B., Winsor Brown, A., Costa, P., Lindvall, M., Port, D., Rus, I., Tesoriero, R. & Zelkowitz, M.V., 2002: What We Have Learned About Fighting Defects, In *Proceedings of 8th International Software Metrics Symposium,* Ottawa, Canada, 249-258.

Sterman, J.D., 2003: Business Dynamics: Systems Thinking and Modeling for a Complex World, Irwin/McGraw-Hill, Chicago, USA.

Taguchi, G., 1981: On-Line Quality Control During Production, Japanese Standard Association, Tokyo, Japan.

Thayer, R.H., 1979: Modeling a Software Engineering Project Management System, Doctoral Dissertation, University of California, USA.

Thayer, R., 1986: Tutorial: Software Engineering Project Management, Computer Society Press of the IEEE, California, USA.

Wickens, C.D. & Hollands, J., 2000: Engineering Psychology and Human Performance, Prentice-Hall, Upper Saddle River, USA.

Williford, J. & Chang, A., 1999: Modeling the FedEx IT Division: A System Dynamics Approach to Strategic IT Planning, *The Journal of Systems and Software*, 46: 203-211.

Wolverton, R.W., 1974: The Cost of Developing Large Scale Software, *IEEE* C-23(6): 615-636.

Det Norske Veritas, 2012: Vurdering av Altinn II-plattformen, Rapport 1239/2011, Rev. 1.1/2012, Oslo, Norge.

ESSU, 2007: Cost Overruns, Delays and Terminations: 105 Outsourced Public Sector ICT Projects, London, Great Britain.

The Standish Group, 2005: The CHAOS Chronicles, West Yarmouth, USA.