UNIVERSITY OF BERGEN

MASTER THESIS

SQuIDS

Software **Qu**ality **I**ssue **D**etection **S**ystem

-

Development of an Eclipse plug-in for automated detection of software maintainability problems

Author:
Lars Alberto Vangsnes CABRERA

Supervisor:
Prof. Solveig BJØRNESTAD

A thesis submitted in fulfilment of the requirements
for the degree of Master

in the

Department of Information Science and Media Studies

June 1, 2016

Keywords:
Software, Quality, Maintainability, CISQ, Eclipse, plug-in,
static source code analysis, usability, performance,
Abstract Syntax Tree (AST), Java

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.

– Sir Charles Antony Richard Hoare

Abstract

Software quality can make a large impact on the cost and speed of development, as well as on what functionality can be delivered in time. Techniques, tools and models exist for measuring and improving software quality. Static code analyzers are programs which can be used to identify quality problems in the source code of software. The *CISQ Specifications for Automated Quality Characteristic Measures* provide a set of measures for automatic analysis, which can be implemented into a static code analyzer. *Maintainability* is a characteristic of software quality, and is one of four characteristics in the CISQ specification. Two implementations of the CISQ specification exist, where one of them, called MUSE, implements the maintainability characteristic. However, neither are available as plug-ins for an Integrated Development Environment (IDE).

In this study, a software artifact named SQuIDS (**S**oftware **Qu**ality **I**ssue **D**etection **S**ystem) was developed as a plug-in for the Eclipse IDE. This was done in order to find out how a static code analyzer can be developed to find maintainability problems in software source code, based on a standard, with focus on correctness, usability and performance. SQuIDS analyzes Java source code, finds maintainability problems defined by the CISQ specification, marks the problems in the source code editor, and provides a software quality score according to the specification.

The software artifact was evaluated by comparing results with the MUSE software (Plösch, Schürz, and Körner, 2015), providing an example of and discussing how maintainability problems can be visualized to the user, and evaluating performance by measuring the time it takes SQuIDS to analyze five existing open-source software projects. The results show that SQuIDS and MUSE find different problems for most of the CISQ measures, that the way SQuIDS visualizes maintainability problems works, but may not be optimal for a larger number of problems, and that although the performance of SQuIDS proved to be lower than desired, makes it usable while developing software.

Further research and development is recommended to improve the correctness, usability and performance of SQuIDS. A method for comparing and verifying the analysis results of SQuIDS and MUSE has been proposed and used for a selection of the CISQ measures. Further verification of the analysis results could therefore be conducted using the method.

SQuIDS is available as an open-source software project on GitHub, and can be installed and used in Eclipse to identify maintainability problems in Java software source code.

Acknowledgements

During the past year, my supervisor Solveig Bjørnestad has kept my thinking straight, and encouraged me to always do my best. Without her expertise, personal library of software engineering, direct speech, pep talks and inspiring random chats, this thesis would never be what it is. For that, I express my deepest gratitude.

I wish to thank my fiancée and our son for taking care of me during stressful days and weeks, and reminding me that no matter the struggle, there is always a warm hug which makes it better.

My good friend Morten Oftedal also deserves my gratitude for suggestions, discussions, debugging, listening to anxiety rants and helping me take my mind off work once in a while by doing something creative.

I thank Severin Schürz for providing me with the raw data results from Plösch, Schürz, and Körner's study, which was imperative in comparing my results. I also thank Bjørnar Tessem for reassuring me that the project seemed feasible, and for guiding me in comparing my results.

Finally, to all my friends who have seen me only sporadically and asked me how I have been doing during the past year, I thank you for thinking of me, and I hope to spend more time with all of you again soon.

Contents

List of Figures

List of Tables

List of Abbreviations

API	**A**pplication **P**rogramming **I**nterface
AST	**A**bstract **S**yntax **T**ree
CBO	**C**oupling **B**etween **O**bject classes
CC	**C**yclomatic **C**omplexity
DSR	**D**esign **S**cience **R**esearch
IDE	**I**ntegrated **D**evelopment **E**nvironment
IEC	**I**nternational **E**lectrotechnical **C**ommission
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
ISO	**I**nternational **S**tandards **O**rganisation
IS	**I**nformation **S**ystem
IT	**I**nformation **T**echnology
JVM	**J**ava **V**irtual **M**achine
LOC or **SLOC**	(**S**oftware) **L**ines **O**f **C**ode
LLOC	**L**ogical **L**ines **O**f **C**ode
PDE	**P**lug-in **D**evelopment **E**nvironment
PLOC	**P**hysical **L**ines **O**f **C**ode
PoC	**P**roof **o**f **C**oncept
regex	**reg**ular **ex**pressions
SQuIDS	**S**oftware **Qu**ality **I**ssue **D**etection **S**ystem
UML	**U**nified **M**odeling **L**anguage
VCS	**V**ersion **C**ontrol **S**ystem
XP	e**X**treme **P**rogramming

Chapter 1

Introduction

An important aspect of software engineering is *software quality*. While there have been many disagreeing definitions of what software quality is, poor software quality has caused companies to lose almost US$60 billions each year. Either the software does not deliver the promised features and functionality, or it has defects which leads to downtime, slow operation or time-consuming maintenance (Bessin, 2004; Pressman, 2010).

Maintainability is a characteristic of software quality, and describes a software program's degree at which it is easy to maintain. In other words, a given software program's maintainability is how easy it is to perform changes on it, thus a measure of how *maintainable* it is (Bourque and Fairley, 2014, c. 10). With the rapidly evolving technology of the 21st century, software is expected to see numerous larger changes and maintenance over its lifecycle, with changed objectives or newly discovered challenges, leading to added or substituted functionality. This requires software to be maintainable, in order to reduce costs and deliver software faster.

The reality is that maintainability of evolving software tends to decrease over time (Bakota et al., 2012, p. 316), and according to Martin (2009), programmers "are *constantly* reading old code as part of the effort to write new code". With this in mind, how do we make software maintainable?

1.1 Techniques for Maintainability

There are numerous techniques available to ensure software maintainability. In software engineering, some of those techniques are (Sommerville, 2011):

- Software design: By providing a rigorous software design as documentation, the software may become more maintainable. In addition, the software may be

developed as maintainable due to the design itself

- Code inspections: Groups of developers review the code, and look for weaknesses. Problems related to maintainability may be detected in these sessions

- Refactoring: Developers improve the source code's structure and organization. This activity can be used to detect maintainability issues

While these techniques can be used to find maintainability problems in the source code, they rely on rigorously defined methods and may require much time and effort in order to keep the software maintainable. Additionally, design documentation is only useful for this purpose if it is frequently maintained, updated and correct in relation to the software. As a remedy for this, can detection of maintainability problems be standardized and automated?

1.2 Static Code Analyzers

Static code analyzers are computer programs which can automatically detect and present violations against defined rules in the software source code, using predefined rules for coding rules and standards. (Broeckman and Notenboom, 2003; Chirilă and Creţu, 2012). These can be used for automatic detection of software quality problems, and thereby maintainability problems. Examples of such analyzers are:

- AWARE
- Checkstyle
- PMD
- SonarQube
- The MUSE software developed by Plösch, Schürz, and Körner (2015)
- Software developed by Satrijandi and Widyani (2015)

The first four analyzers are available as plug-ins for the Eclipse IDE. However, AWARE is focused on *reliability*, which is another characteristic of software quality, and the only available downloads are older versions which are no longer supported. In addition, these four analyzers are not based on any standard. On the other hand, the last two software programs are based on the CISQ Specifications for Automated Quality Characteristic Measures, which consist of descriptions of metrics for the four software quality characteristics *reliability*, *performance efficiency*, *security* and *maintainability*. MUSE, which is developed by Plösch, Schürz, and Körner (2015), implements the maintainability characteristic, whereas the software developed by

Satrijandi and Widyani (2015) implements the performance efficiency characteristic. These programs are however not available for download.

1.3 Research Questions

This brings us to the problem: No static code analyzer is based on an existing standard, can automatically detect maintainability problems *and* is available as a plug-in for an IDE. Therefore, this research project will intend to create such a software as a proof of concept. Additional goals are to find out if such a software could be expected to have an acceptable performance, so that software developers would be willing to use it during their day-to-day work, and how problems that are found may be displayed to the user. The overarching research question for this project is consequently:

How can a static code analyzer for maintainability, based on an existing standard, be developed as a plug-in for an IDE, such as Eclipse, with focus on correctness, performance and usability?

This leads to the following research questions:

Q1 - How can a static code analyzer be developed as an Eclipse plug-in which, based on a standard, detects maintainability problems in a software project?

Q2 - How can such an implementation visualize maintainability problems in a software project to the user?

Q3 - Can such an implementation give a performance which makes it usable during development?

1.4 Organization of the Thesis

This thesis has started with a presentation of the background and a description of the problem. In Chapter 2, relevant theory and related research will be presented. Following, the research methods and the research design of this project are explained in Chapter 3. Chapter 4 presents the development process of the software, and the results are presented in Chapter 5. Chapter 6 discusses the research results in relation to research methods and relevant theory, answers the research questions, and challenges the validity of the research results. Chapter 7 presents conclusions and further research.

Chapter 2

Theory

This chapter introduces important concepts in *software engineering* and *software quality analysis* and presents related research.

2.1 Software Engineering

Software Engineering implies the use of engineering in the management of software. It is a systematic, disciplined and quantifiable approach to software development, operation and maintenance, with techniques for software specification, design and evolution (Sommerville, 2011, p. 5; Bourque and Fairley, 2014, p. xxi). The following subsections describe some of these techniques.

2.1.1 Software Requirements

A software requirement is a description of a feature which the software must comply with to solve a real world problem. The requirements of a software system are typically generated from various people in an organization, and people who are somehow involved with the operational environment of the software. Requirements are identified as either functional (individual features) or nonfunctional (system level requirements) (Bourque and Fairley, 2014).

Bourque and Fairley (2014) define functional requirements as functions which the system must execute, whereas nonfunctional requirements are constraints on the software, or quality requirements. Further division of nonfunctional requirements include requirements on performance, maintainability, safety, reliability, security and interoperability.

There are a number of techniques for eliciting (sometimes referred to as discovering) requirements, such as the following (Sommerville, 2011, c. 4, Bourque and Fairley, 2014, c. 1):

- Interviews
- Scenarios
- Use cases
- User stories
- Prototypes

2.1.2 Scenarios

One technique for requirement elicitation is to create *scenarios*. A scenario is an example of how a system may be interacted with. An outline of the interaction is written first, and more details are added during the elicitation process. The information in the description of a scenario may contain:

- The system's and users' expectations when the interaction starts
- The normal event flow
- Possible errors and handling of them
- Information of other possibly simultaneous activities
- The state of the system at the end of the interaction

Multiple scenarios are created to cover different interactions, levels and details of the system (Sommerville, 2011, p. 105).

2.1.3 User Stories

Writing *user stories* is another requirements elicitation (i.e. collection) technique used in adaptive development methods such as agile methods. In contrast to detailed, low-level descriptions of requirements, a user story is a short, high-level description written with terms which are understandable for the customer (Bourque and Fairley, 2014, p. 38; Wells, 1999c).

User stories are typically written in the following pattern: *As a <role>, I want <goal/desire> so that <benefit>*. With this approach, which will be used in the project, it contains only the relevant information needed for a developer to implement the story, and to estimate how large the workload will be. User stories are meant to reduce the chances that a requirement becomes invalid as the project progresses (Bourque and Fairley, 2014, p. 38).

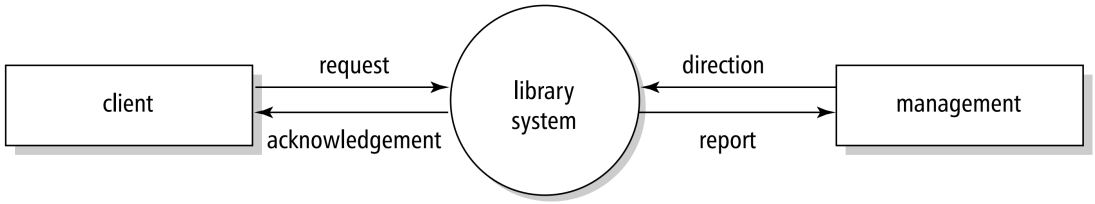

FIGURE 2.1: Example of a library system context diagram, from Vliet (2007).

2.1.4 Software Design

Software design is a creative process where requirements are turned into system components and the relationships between them (Sommerville, 2011, p. 194). Techniques for designing software include, amongst others, context diagrams, UML class diagrams and process diagrams (Vliet, 2007, p. 343). UML (Unified Modeling Language) is a collection of graphical elements used to notate the description and design of software. It is an open standard regulated by the Object Management Group (OMG), and is mainly intended for designing object-oriented software.

2.1.4.1 Context Diagrams

A *context diagram* is a high-level model of the intended context of the system. It is used to provide a description of the interaction between the system and the environment, and are drawn as data flow diagrams, which contain specific notations for data stores, data flows, processes and external entities. Contrary to data flow diagrams, context diagrams may contain only a single process, which indicates the system in question. (Vliet, 2007, p. 349). An example context diagram is illustrated in Fig. 2.1, depicting a library system.

2.1.4.2 Class Diagrams

The classes of objects in a system and the relationships between them can be modeled with a UML *class diagram*. In UML, classes are displayed as nodes (boxes), and relationships as edges (arrows). Classes may be drawn as boxes with one compartment for name, one for attributes (fields) and one for operations (methods). Class diagrams are made to describe the static structure of a system (Fowler, 2004, p. 35; Vliet, 2007). Fig. 2.2 illustrates an example of a small class diagram.

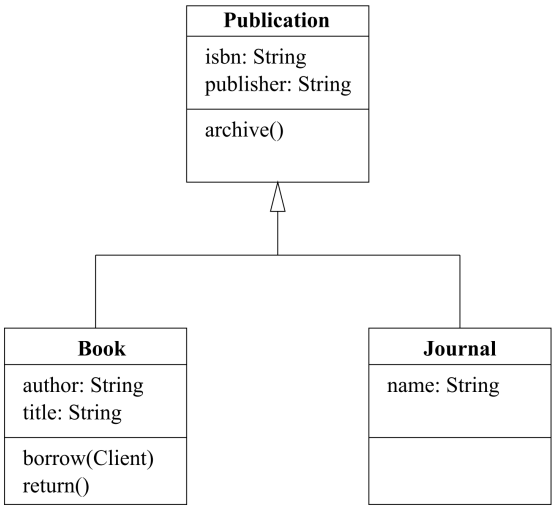

FIGURE 2.2: Example of UML a class diagram, from Vliet (2007).

2.1.4.3 BPMN Diagrams

The Business Process Modeling and Notation (BPMN) is a notation standard used to create process diagrams of information systems, and is intended to communicate to a wide audience (Object Management Group, 2011). Fig. 2.3 is an example BPMN diagram of a patient-doctor collaboration process.

2.1.5 Software Response Time

Users have certain expectations when it comes to the response time of software. Nielsen (1993) specifies three human perceptional time limits for software systems:

- **0.1 second** - the maximum time it can take for the user to feel an *instant reaction* from the system. The result is the only feedback necessary
- **1.0 second** - The user no longer feels that the system provides instant reaction, but keeps the *flow of thought* uninterrupted up until this limit. Feedback other than the result should normally not be necessary
- **10 seconds** - The *user's attention* on the dialog is kept until this limit. Beyond this limit, users should be provided with feedback which indicates how much time is remaining to complete the operation, so they can perform other tasks in the meantime. If the response time is highly variable, feedback is especially important to give during the operation, as the users will not know when to expect it to be done

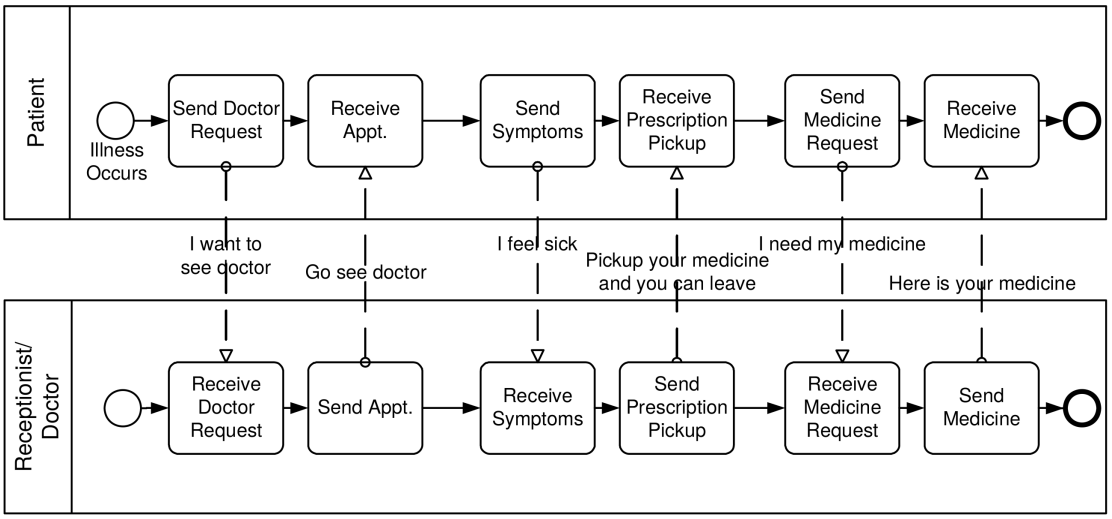

FIGURE 2.3: Example of a BPMN diagram (Object Management Group, 2011, p. 25).

A percentage of the operation performed, a *percentage-done indicator*, should be provided when the operation takes more than 10 seconds. If the amount of work in an operation cannot be calculated on beforehand, this may not be possible, but an indication of what has been done can still be given. For example, in an application which performs a complex algorithm with certain steps which are iterated an unknown number of times to achieve an optimal result (e.g. genetic algorithm), the number of iterations and the state of the result can be provided to the user per iteration (Nielsen, 1993).

2.1.6 Integrated Development Environment

After setting the requirements and designing the software, the software is often developed in an *Integrated Development Environment (IDE)*. An IDE is a system of development tools used by programmers for creating software. Beyond a source code editor, an IDE typically features compilation and error reporting directly in the editor, version control system integration, tools for building, testing and debugging the code, abstract representations of programs, and automatic transformation and generation of code for refactoring (Bourque and Fairley, 2014, c. 3, p. 12). An example is the Eclipse IDE, which is used to develop software in many programming languages, though usually Java (The Eclipse Foundation, 2016).

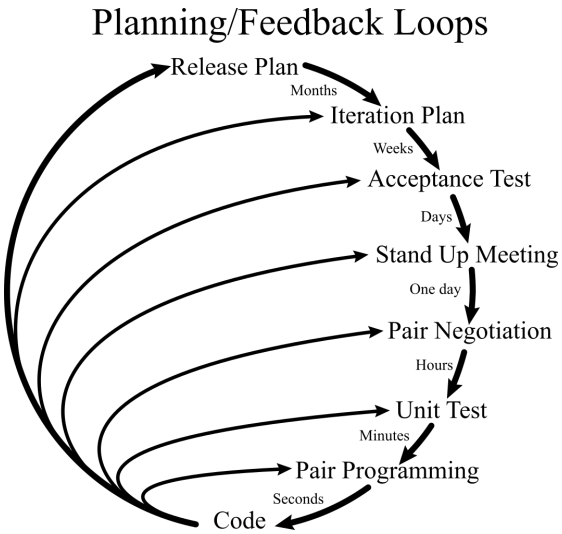

FIGURE 2.4: eXtreme Programming Project (Wikimedia Commons, 2001).

2.1.7 eXtreme Programming

Software development can be managed using a defined development method. eXtreme Programming (XP) is an agile software development method which focuses on communication, simplicity, feedback, respect, and courage. Daily face-to-face communication and having the development team work together on all parts of development is intended create the best solution to the problem. Developers should simplify the software by only focusing on what is needed. Working software should always be delivered in order to provide and retrieve feedback to and from the customer. Mutual respect between development team members, the customer and management is important, and the development team must both accept responsibility for and receive authority over their code. Developers should be honest about progress and problems, and not be afraid to adapt to changes (Wells, 2009).

In XP, development is divided into iterations (see Fig. 2.4), consisting of planning, coding, testing and evaluation. Iterations last optimally one week and should not exceed three weeks, and the length should be kept constant throughout development (Wells, 1999c). User stories are written (see Section 2.1.3), and make the basis of an iteration. An XP iteration planning meeting consists of selecting priority user stories and failed acceptance tests (user stories which did not pass acceptance) to include in the iteration, dividing them into programming tasks, and estimating implementation time for tasks (Wells, 1999a). User stories are estimated in story points, which are used to estimate how much work can be done during an iteration, and to calculate the project velocity at the end of an iteration. This is done by simply adding up the story points completed during an iteration. The project velocity can be used for estimating

how much work can be expected during future iterations (Wells, 1999b).

2.1.8 Unit Testing

A software program's functional behavior can be tested and evaluated by creating unit tests. Software may be decomposed into units of functions, and unit tests are written to test those units, to ensure that the program's functionality behaves as intended. Creating unit tests requires specification of input values for each unit, and asserting the expected behavior (Sen, Marinov, and Agha, 2005, p. 263). Snippet 2.1 is an example of a unit test case for the method `add()` in a Java class called `Money`, using the JUnit framework.

Snippet 2.1: Example unit test method for a Java program (adapted from junit.org).

```
1  @Test
2  public void simpleAdd() {
3      // Two Money objects are created
4      Money m12CHF= new Money(12, "CHF");
5      Money m14CHF= new Money(14, "CHF");
6
7      // The expected result is specified.
8      Money expected= new Money(26, "CHF");
9
10     // The actual result is created.
11     // The m14CHF object is the unit input value.
12     // The result object is the unit output value.
13     Money result= m12CHF.add(m14CHF);
14
15     // It is assured that the expected result is equal to the actual result.
16     // If not, JUnit reports the test as failed.
17     assertTrue(expected.equals(result));
18 }
```

In XP (see Section 2.1.7), a programmer should write the unit test before the unit itself. This is a practice also known as the Test First Principle (TFP), and aims to save development time, by knowing when the code works as intended (Wells, 2000).

2.1.8.1 Black Box vs. White Box

One manner of unit testing is known as functional testing, or Black Box, which are test design techniques where the system is subjected to input, and the output is analyzed according to the expected behavior. In contrast to White Box techniques, where the focus is on the knowledge of the system's internal structure, Black Box techniques only

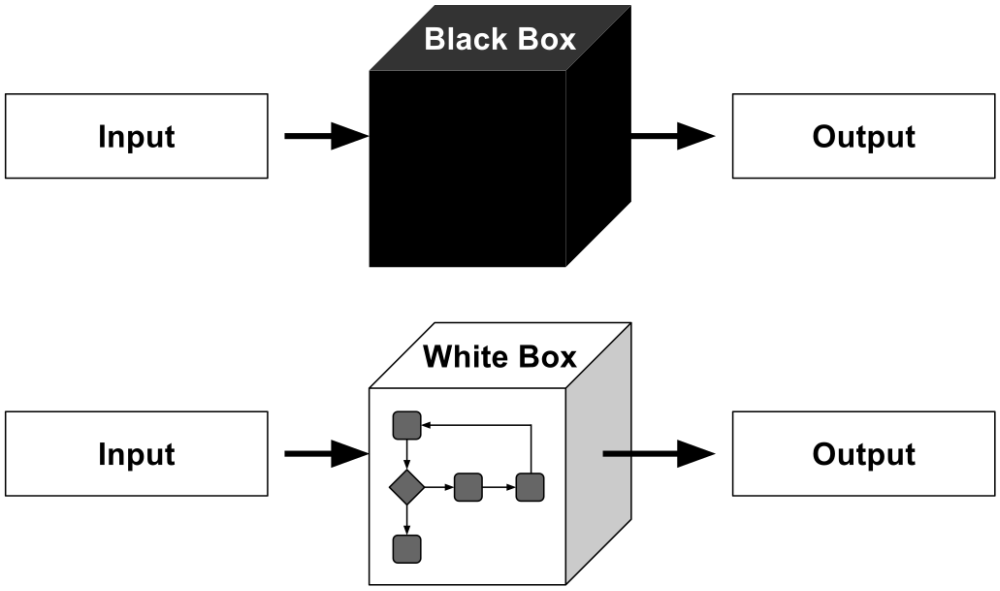

FIGURE 2.5: Black Box vs. White Box testing (Adopted from softwaretestinggenius.com).

focus on expected output from a provided input (Broeckman and Notenboom, 2003, p. 115). The difference is illustrated in Fig. 2.5.

2.1.8.2 Boundary Value Analysis

The principle of a *boundary value analysis* test technique is to detect programming errors in logic related to boundaries. A typical mistake is using the wrong comparison operator, such as < (less than), when the correct operator should have been <= (less than or equal to). The boundary values should therefore be tested with two test cases per boundary, surrounding the boundary values (Broeckman and Notenboom, 2003, p. 119).

For example, in a computer game, where the health points of a `Player` object can be altered using a method called `setHealth(int health)`, there is an infinite amount of possible input values[1]. However, the `Player` can only have a minimum of 0, and maximum of 100 health points. Therefore, the boundary values for `setHeatlh()` are 0 and 100:

Example 2.1

$$0 \leq health \leq 100$$

The test cases for the `setHealth()` method should therefore be:

[1] In Java, however, the smallest possible value for an integer (`int`) is -2147483648, and the maximum 2147483647.

1. assert that health is set to 0 when input value is -1
2. assert that health is set to 0 when input value is 0
3. assert that health is set to 100 when input value is 100
4. assert that health is set to 100 when input value is 101

Additionally, a third test case per boundary (two extra test cases in total) can be added to ensure thoroughness, testing values which are just within the equivalence partition of the boundary value (Broeckman and Notenboom, 2003, p. 119):

5. assert that health is 1 when input value is 1
6. assert that health is 99 when input value is 99

These additions can detect an error where the programmer erroneously wrote `0 == health` (`health` equal to `0`) instead of `0 <= health` (`health` less than or equal to `0`), and vice versa for `100`.

2.1.8.3 Test Formality

A test design can have formal or informal rules on how tests are written. A formal test design specifies strict rules on which test cases and how the test cases should be written, making sure that the tester does not forget important aspects to test. Whereas this limits the test quality to the quality of the test specification, an informal test design allows the tester to be creative, and not rely on the quality of the test specification. The disadvantage of an informal approach is that the test coverage cannot be derived from the test design, leaving the possibility that the tester forgot something important (Broeckman and Notenboom, 2003, p. 120).

2.1.8.4 Unit Test Coverage

Two techniques for evaluating how much of the code the unit tests cover in a software are measuring *code coverage* and performing *mutation tests* (sometimes referred to as mutation coverage). Code coverage is a measure of how much of the software source code is covered by unit tests. One version of code coverage is *line coverage*, where the number of covered lines in the source code is the unit of measurement (Rojas et al., 2015, p. 97). Mutation testing is a method of creating mutations of the source code with logical errors to see if the errors are caught by the unit tests or not (Li, Praphamontripong, and Offutt, 2009, p. 222). An existing tool which performs both line coverage measurement and mutation testing is the software PITEST (sometimes abbreviated as PIT).

2.1.9 Software Engineering Summary

Applying software engineering techniques to software development provides a structured approach to specifying the requirements of the software, designing it and developing it. The techniques ensure that the scope and intention of the software is thoroughly defined and able to be evaluated, and that the software is developed efficiently.

2.2 Software Quality

As stated in Chapter 1, *Software quality* is an ambiguous term, and it has seen many definitions over the years. One definition, cited from the ISO/IEC 25010:2011 standard, is the "capability of software product to satisfy stated and implied needs under specified conditions" (Bourque and Fairley, 2014, c. 10, p. 1). A more general definition is "An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it" (Pressman, 2010, p. 400).

Bourque and Fairley (2014) state that software quality is achieved by conforming to all the requirements of the software, and refer to recent definitions, which underline that the quality of the software depends on is requirements.

From an ethical point of view, there is an expectation that software engineers have a cultural relationship to software quality. This includes the knowledge of the relationship between quality, time and cost, and the trade-offs between them (Bourque and Fairley, 2014, c. 10, pp. 2-3).

2.2.1 Models and Quality Characteristics

In order to discuss, plan and rate software quality, several models of software quality characteristics have been created, each with its own taxonomy of quality characteristics. One of these models is the model standardized in ISO/IEC 25010, which consists of eight software product quality characteristics, illustrated in Fig. 2.6 (Bourque and Fairley, 2014, c. 10, p. 3). The ISO/IEC 25010 standard is part of the ISO/IEC 25000 series *System and Software Quality Requirements and Evaluation* (SQuaRE) from 2011. The series replaces the ISO 9126 standards from 1991 (CISQ, 2012, p. 5).

FIGURE 2.6: Software Product Quality Characteristics from ISO/IEC 25010 (Iso25000.com, 2015).

2.3 Software Maintainability

One of the software quality characteristics in the SQuaRE model (see Section 2.2.1) is *maintainability*. As with the term software quality, ISO/IEC has also provided various definitions of software maintainability over the years (Bourque and Fairley, 2014, c. 5, p. 5; IEEE Standards Association, 2006, p. 3; IEEE Standards Association, 2010, p. 204; CISQ, 2012, p. 9; ISO/IEC, 2011, p. 9). The definitions are similar in terms of semantics. However, the latest definition is alone in including *the intended maintainers* of the software as a specification: "The degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers." (CISQ, 2012, p. 9; ISO/IEC, 2011).

In other words, the maintainability of a software is a rating of how little effort is required by its intended maintainers to perform modifications on it, i.e. how maintainable the software is.

2.3.1 Maintainability in Use

It is often difficult for developers to achieve maintainability while creating software, because other activities require their attention and are prioritized to meet deadlines. Additionally, software documentation and test environments may be lacking due to disregard of the maintainer's requirements. As a result, the software becomes even harder to maintain. On the other hand, matured and systematic tools and processes can increase the maintainability of the software by aiding the developers (Bourque and Fairley, 2014, c. 5, p. 5).

Bourque and Fairley (2014) list the following measures for software maintenance:

- **Analyzability** - measures of the maintainer's effort or resources expended in

trying either to diagnose deficiencies or causes of failure or to identify parts that need to be modified.

- **Changeability** - measures of the maintainer's effort associated with implementing a specified modification.
- **Stability** - measures of the unexpected behavior of software, including that encountered during testing.
- **Testability** - measures of the maintainer's and users' effort in trying to test the modified software.

In addition, Bourque and Fairley mention the size, complexity and understandability of the software.

2.3.2 Maintainability Metrics

There are numerous established metrics for measuring software maintainability[2]. For instance, Sjøberg, Anda, and Mockus (2012) list the following:

- Lines Of Code (LOC): A count of the total number of code lines in the software (see Section 2.4.5)
- Number of comments
- Cyclomatic complexity (CC): A measure of the complexity of a single software module (see Section 2.4.6)
- Halstead's Volume: A measure of the size of the software

A multitude of other metrics exist, such as the metrics in the taxonomy of software maintainability metrics created by Oman and Hagemeister (1992), the object-oriented software maintainability metrics categorized by Saraiva, Soares, and Castor (2013), and the metrics for automatic maintainability measurement in the CISQ Specifications for Automated Quality Characteristic Measures (CISQ, 2012). Common between these works are metrics concerning complexity and size, while their differences relate to the objective and scope of the research, and detail of the metrics.

Oman and Hagemeister (1992) define 49 metrics, create a taxonomy of them, and propose a method for calculating a maintainability index of a software, based on the metrics. Saraiva, Soares, and Castor (2013) categorize 570 metrics, but do not list all of them, and give no definitions, as the objective of their paper is simply to categorize known metrics in order to make the process of building a catalog of object-oriented

[2] Although, Sommerville (2011) claims there are no metrics for maintainability.

software maintainability metrics accurate and reliable. CISQ (2012) list only 21 metrics for software maintainability, and provide little or no definition of them. However, the goal of the specification is to provide a list of metrics which can be automatically measured, and where violations of the metrics have a high severity of impact on the software, i.e. the level of harm a violation causes on maintenance or operation of the software.

Since many of the metrics require system- and programming language-specific thresholds (e.g. how many LOC are too many) and weights (e.g. the impact of the metrics on the Maintainability Index (Oman and Hagemeister, 1992, p. 343)), the use of *fuzzy logic*-based algorithms for setting them have been proposed (Dahiya, Chhabra, and Kumar, 2007; Chen and Liu, 2009).

2.3.3 Related Research on Maintainability Metrics

in Using Metrics to Evaluate Software System Maintainability, Coleman et al. (1994) intend to demonstrate automated software maintainability analysis and how it can be applied in decision-making related to software.

2.3.3.1 Tools

The researchers used the HPMAS software maintainability assessment system, a hierarchical multidimensional assessment model created by HP (Hewlett Packard). The model was run on 11 industrial software systems.

2.3.3.2 Model

The article refers to a hierarchical model by Oman and Hagemeister, which divides maintainability into three underlying dimensions or attributes (Coleman et al., 1994, p. 45):

1. The *control structure*, which includes characteristics pertaining to the way the program or system is decomposed into algorithms

2. The *information structure*, which includes characteristics pertaining to the choice and use of data structure and dataflow techniques

3. *Typography, naming and commenting*, which includes characteristics pertaining to the typographic layout, naming and commenting of code

2.3.3.3 Conclusions

Results from their analysis conform with "engineers' intuition about the maintainability of the (sub)system components". However, the automated analysis provided a deeper understanding, and supported the engineer's opinions with additional data. Automated maintainability analysis can be conducted at all levels of the system; at component, sub-system and whole system level (Coleman et al., 1994, p. 49).

2.4 Software Quality Analysis

This section contains descriptions of concepts and tools used for analyzing software in order to assess its quality, in addition to related research.

2.4.1 Abstract Syntax Tree and Tokens

An Abstract Syntax Tree (AST) is a representation of code as a tree, where each tree node is a construct of the source language, containing zero or more children nodes (see in Fig. 2.7). Unnecessary syntactic details like punctuation or whitespace is omitted, whereas order, structure and types are gathered as *tokens* (keywords) and maintained in the tree format (Jones, 2003, p. 2; Lucas, 2006; Grune, 2012). Examples of such tokens in the Java programming language are listed in Table 2.1.

TABLE 2.1: Examples of Java tokens.

Token	Explanation
`int`	Primitive type
`"Hello!"`	String literal
`1`	Integer literal
`public`	Access modifier

While ASTs and tokens are concepts often used in compiler software, they provide a structured view of the relationships between the constructs in a software system, which can be used to answer questions such as:

- "how many new variables are introduced in this function?"
  or
- "Is there a loop within a loop, which is within another loop?"

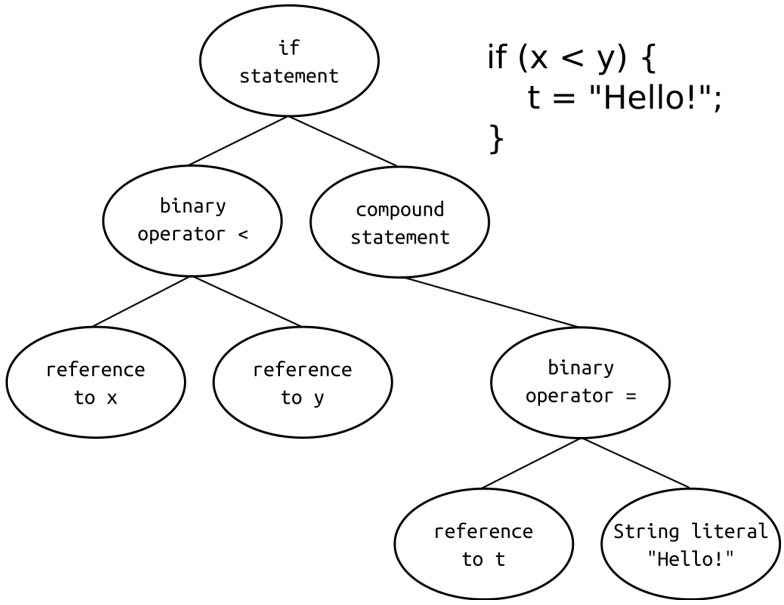

FIGURE 2.7: Example of an Abstract Syntax Tree adopted from an article at ics.com

2.4.1.1 Visitor Pattern

A typical technique of traversing ASTs is using the programming pattern *Visitor* (Rashid and Pottier, 2012, p. 324; Nystrom, Clarkson, and Myers, 2003, p. 142). A visitor is a class which performs an operation on another object. The object accepts the visitor with an `accept(Visitor visitor)` method, which in turn calls either of the visitor's `visit(...)` methods, depending on which type the object is, by the principle of *method overloading*. For example, if the visited object is an instance of the class `ClassA`, the visitor's `visit(ClassA classA)` method, and the `visit(ClassB classB)` method if the visited object is a `ClassB` instance (see Fig. 2.8). This makes type-casting unnecessary, as the static type checking is performed automatically, and type-specific operations can be performed by the visitor (Kerievsky, 2004).

2.4.2 Architectural Layers

A software system may be divided into logical layers of components, which helps sorting the different responsibilities of the components (Mitra, 2008; MSDN, 2009; Xu and Wan, 2015). It is recommended to maintain a structured layer architecture in order to keep the software maintainable (Mitra, 2008; CISQ, 2012, p. 25; Fowler, 2015; MSDN, 2009).

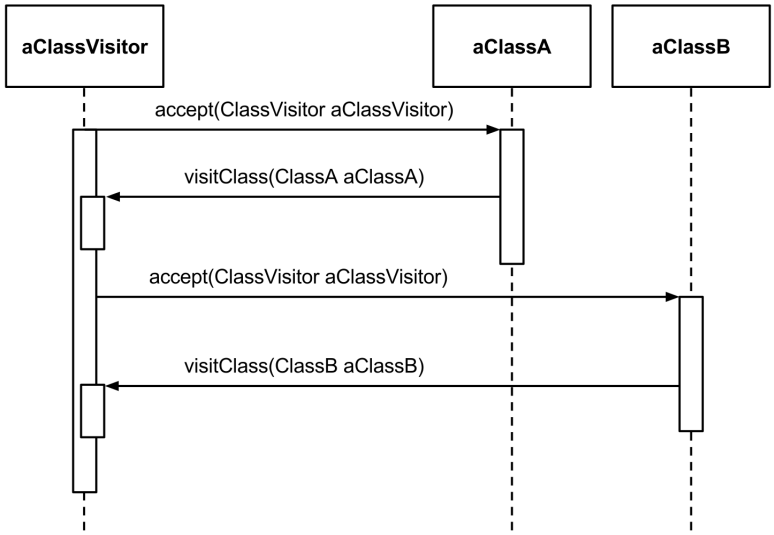

FIGURE 2.8: Example diagram of a visitor pattern (adapted from Kerievsky (2004))

A popular layer architecture is the Three-Tier Architecture model, (see Fig. 2.9). This model divides software in three layers (Fowler, 2015; Xu and Wan, 2015):

1. presentation / interface layer
2. domain / business layer
3. data / persistence layer

Some software engineering scholars and professionals distinguish between the terms *layer* and *tier*, while others refer to them synonymously. MSDN states that there is an important difference between the two – that layers are logical groupings of components, whereas tiers refer to the physical locations of components on separate machines or networks (MSDN, 2009). Xu and Wan refer to the tiers in the Three-Tier Architecture model as not necessarily being physical locations, but rather logical layers (Xu and Wan, 2015, p. 1).

Fowler (2015) and Mitra (2008) emphasize the importance of keeping the dependencies between layers from above to below – that a layer should only access layers below, and not above. Mitra also distinguishes between horizontal and vertical layers (see Fig. 2.10), where vertical layers may depend on, or be dependable by, one or more horizontal layers.

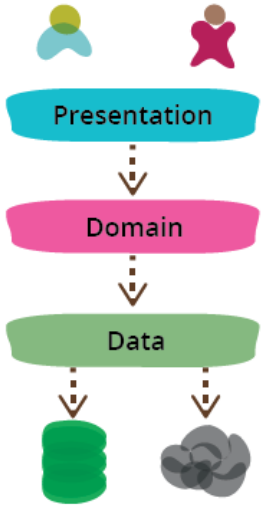

FIGURE 2.9: Presentation-Domain-Data Layering (Fowler, 2015).

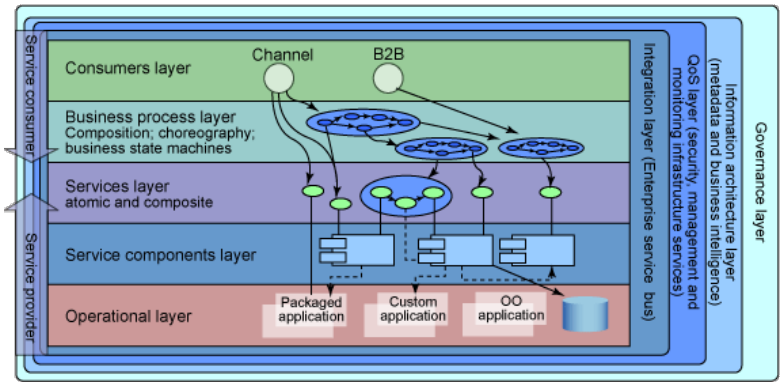

FIGURE 2.10: Layered architecture view depicting an SOA reference architecture (Mitra, 2008).

2.4.3 Coupling

One of the measures of software maintainability often referred to is *coupling*, where *Coupling Between Object classes (CBO)* from Chidamber and Kemerer (1994) is a well-known metric (Muthanna et al., 2000, p. 1; Lincke, Lundberg, and Löwe, 2008, p. 134; Shen, Zhang, and Zhao, 2008, p. 1; Saraiva, Soares, and Castor, 2013, p. 85; Plösch, Schürz, and Körner, 2015, p. 328).

The principle of coupling is the interdependency between parts of a system design, e.g. objects in an object-oriented system design. An object can be represented in the following manner (Chidamber and Kemerer, 1994, p. 479):

Definition 2.1 ***Object***

$$X = \langle x, p(x) \rangle$$

where z is the substantial individual and $p(x)$ is the finite collection of its properties.

Formally, object coupling is defined as the following (Chidamber and Kemerer, 1994, p. 479):

Definition 2.2 ***Coupling***

Let $X = \langle x, p(x) \rangle$ and $Y = \langle y, p(y) \rangle$ be two objects.

$$function\ p(x) = \{M_X\} \cup \{I_X\}$$

$$function\ p(y) = \{M_Y\} \cup \{I_Y\}$$

where $\{M_i\}$ is the set of methods and $\{I_i\}$ is the set of instance variables of object i. Any action by $\{M_X\}$ on $\{M_Y\}$ (call to a method in object Y) or $\{I_Y\}$ (call to an instance variable in Y), or, vice versa, any action by $\{M_Y\}$ on $\{M_X\}$ or $\{I_X\}$ constitutes coupling.

Chidamber and Kemerer's CBO measures the coupling of a class by counting the number of other classes to which it is coupled (Chidamber and Kemerer, 1994, p. 486). Using Definition 2.1 and Definition 2.2, an example of the CBO of the class of object X is:

Example 2.2 ***CBO***

Let $Z = \langle z, p(z) \rangle$ be another object.

$$function\ p(z) = \{M_Z\} \cup \{I_Z\}$$

where there is an action by $\{M_X\}$ on $\{M_Y\}$ and $\{I_Z\}$, gives the class of the object X a CBO of 2.

2.4.4 Instructions

Oman and Hagemeister (1992) and CISQ (2012) refer to the term *instruction* in relation to measures for software maintainability. In computer science, an instruction is a method of passing from one *state* of a computer to another. A state is any combination of values in the elements of the computer's memory. Therefore, an instruction alters the computer's memory, thus changing its state. Following is a formal definition of instructions (Maurer, 1966, p. 227):

Definition 2.3 *Let M and B be finite sets.*
Let $\mathcal{S}$ be the sets of all mappings $S : M \to B$.
Let $\mathfrak{g}$ be a set of maps $I : \mathcal{S} \to \mathcal{S}$.

then the 4-tuple $(M, B, \mathcal{S}, \mathfrak{g})$ is a finite complete computer. *The set M is the* memory *of the computer; the set B is called the* base set*; the members $S \subset \mathcal{S}$ are called the* states*; and the members $I \in \mathfrak{g}$ are called the* instructions.

2.4.5 Lines of Code

A measure of the size of software which is recurring in scientific literature concerning maintainability metrics is Lines of Code (LOC) (e.g. Oman and Hagemeister, 1992; Lincke, Lundberg, and Löwe, 2008; Bagheri and Gasevic, 2011; CISQ, 2012). It is sometimes referred to as Software Lines of Code (SLOC) or Kilo Lines of Code (KLOC), as in thousands of LOCs.

Nguyen et al. (2007) conclude that there is no standardized definition of how to count lines of code, and lists two measures of LOC as the most popular and accepted: *physical LOC* (PLOC) and *logical LOC*. While physical LOC counts all lines except lines which are blank or consist of only comments, logical LOC counts statements, which is independent of the physical format of the code (Nguyen et al., 2007, p. 5). The following example of two code snippets are adopted from Nguyen et al. (2007):

Snippet 2.2: With brackets and indentation

```
1 // Check if a number is positive
2 if (x > 0) {
3     System.out.println("x is positive");
4 }
```

and

TABLE 2.2: Results from CodeCounter™ analysis of the snippets.

Code	Visible LOC	Physical LOC	Logical LOC
Snippet 2.2	4	3	2
Snippet 2.3	4	1	2

Snippet 2.3: Without brackets and indentation

```
1 // Check if a number is positive
2 if (x > 0) System.out.println("x is positive");
```

These code blocks perform the same tasks, but have a different number of visible lines, and may therefore produce a different LOC value depending on which LOC counting tool is used. Nguyen et al. (2007) suggest the USC CodeCount™ measure as an LOC counting standard. USC CodeCount™ counts the number of statements in the source code, where statements are considered as blocks of code which perform some action at runtime or direct compilers while compiling. As an example, the CodeCounter™ tool was downloaded and used to analyze the two snippets. According to the tool, Snippet 2.2 has a physical LOC of 3, whereas Snippet 2.3 has only 1, and both have a logical LOC of 2 (see Table 2.2).

2.4.6 Cyclomatic Complexity

The McCabe Cyclomatic Complexity (CC) metric (1976) is a measure of the complexity of a single software module, and is referenced in multiple maintainability related articles (e.g. Bagheri and Gasevic, 2011, p 586; Saraiva, Soares, and Castor, 2013, p. 85; Plösch, Schürz, and Körner, 2015, p. 328).

The metric analyzes the software's *control flow graph* – a description of the logic structure of a software module. A software module is considered as a function, subroutine or any design component featuring a call/return mechanism. The control flow graph contains nodes and edges, where the nodes are computational statements or expressions, and the edges are transfers of control between the nodes (McCabe, Wallace, and Watson, 1996, p. 7). The complexity of a software module can be calculated by counting the number of edges and subtracting from it the number of nodes and adding 2. Formally, the following definition applies (McCabe, Wallace, and Watson, 1996, p. 10):

Definition 2.4 *Let e be the number of edges and let n be the number of nodes in a software module.*

The CC $v(G)$ of a control flow graph G is defined as:

$$function\ v(G) = e - n + 2$$

A simpler calculation is done by simply counting the number of decision predicates and adding 1: $v(G) = p + 1$. Decision predicates are considered as nodes with exactly two edges emerging from them (McCabe, Wallace, and Watson, 1996, p. 23). Below is a simple example method:

Snippet 2.4: Example method with cyclomatic complexity of

```
1  public static int max(int x, int y) {
2      if (x > y) {
3          return x;
4      }
5      else if (y > x) {
6          return y;
7      }
8      else {
9          return x;
10     }
11 }
```

This method in Snippet 2.4 contains 2 decision predicates: `if (x > y)` and `else if (y > x)`. Therefore, the CC of the above software module is $v(G) = p + 1 = 2 + 1 = 3$. By using the more complex calculation on the same example, the CC is the same: $v(G) = e - n + 2 = 11 - 10 + 2 = 3$.

2.4.7 CISQ Specifications for Automated Quality Characteristic Measures

The Consortium for IT Software Quality (CISQ) released a paper in 2012 describing "... a specification for automating the measurement of four Software Quality Characteristics - Reliability, Performance Efficiency, Security, and Maintainability" (CISQ, 2012, p. 3). Each characteristic includes a number of measures for calculating violations of rules of good architecture and coding practice related to that characteristic. CISQ is an industry-led initiative comprised of IT industry professionals. It was launched in 2009 by the Carnegie Mellon Software Engineering Institute (SEI) and the Object Management Group (OMG) (see Software Engineering Institute, 2009). Its goal is to introduce a computable metrics standard to use in measuring software quality and size (CISQ - Consortium for IT Software Quality).

2.4.7.1 ISO/IEC 25010

The characteristics in the CISQ specification confine to the ISO/IEC 25010 standard for system and software quality models (see Section 2.2.1). Where ISO/IEC 25010 defines eight Software Quality Characteristics, four of them are included in the CISQ specification, highlighted in orange in Fig. 2.11 (CISQ, 2012, p. 7).

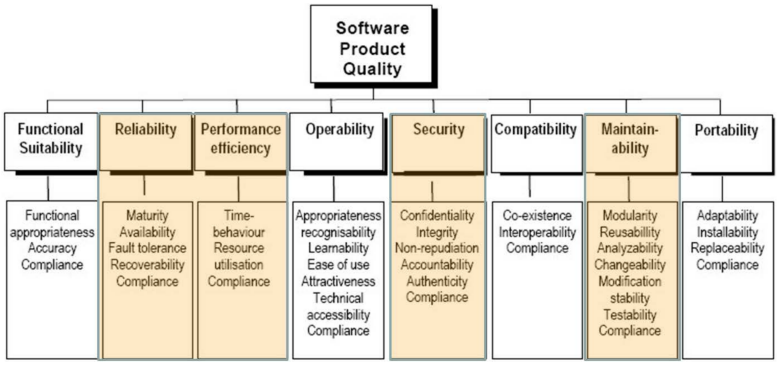

FIGURE 2.11: Software Product Quality Characteristics from ISO/IEC 25010 (CISQ, 2012, p. 7)

TABLE 2.3: Format of the CISQ Maintainability Measure Elements (CISQ, 2012, pp. 25–29).

Issue	Quality Rule	Quality Measure
Issue 1: In a layered architecture functions should be strictly allocated to layers and maintain a strict hierarchy of calling between layers (utility layer excepted).	**Rule 1**: Functions only communicate (exchange data) with functions belonging to an adjacent layer. Functions do not directly exchange data with functions that are not in adjacent layers (no layer skipping/bridging).	**Measure 1**: # of functions that span layers.
		Measure 2: # of layer-skipping calls.
	Rule 2: Avoid too many horizontal layers.	**Measure 3**: # of layers (threshold $4 \leq \#Layers \leq 8$).

The CISQ specification extends the four characteristics "...to the detail required to create measures for each Quality Characteristic that can be computed from statically analyzing the source code." (CISQ, 2012, p. 6), and divides them into issues, rules and measures (see Table 2.3). The results from the measures return a characteristic "score" to identify the problem areas of the software design. The rules are standardized (not specific for any programming language), to make an unambiguous quality score.

2.4.7.2 Maintainability Measures

21 measures for maintainability are included in the specification, listed in Table 2.4.

TABLE 2.4: CISQ Maintainability Measures

CISQ Measure	Description
M01	# of functions that span layers.
M02	# of layer-skipping calls.
M03	# of layers (threshold $4 \leq \#Layers \leq 8$).
M04	# of files that contain $100+$ consecutive duplicate tokens.
M05	# of unreachable functions.
M06	# of classes with inheritance levels ≥ 7.
M07	# of classes with ≥ 10 children.
M08	# of instances of multiple inheritance of concrete implementation classes (threshold > 1).
M09	# of methods that are directly using fields from other classes.
M10	# of variables declared public.
M11	# of functions that have a fan-out ≥ 10.
M12	# of objects with coupling > 7.
M13	# of cyclic calls between packages.
M14	# of functions with $> 2\%$ commented out instructions.
M15	# files with > 1000 LOC.
M16	# of instances of indexes modified within its loop.
M17	# of GO TOs, CONTINUE, and BREAK outside the switch.
M18	# of functions with cyclomatic complexity $\geq$ a language specific threshold (table to be inserted).
M19	# of methods with ≥ 7 data or file operations.
M20	# of functions passing ≥ 7 parameters.
M21	# of hard coded literals except (-1, 0, 1, 2, or literals initializing static or constant variables).

2.4.7.3 Compliance

The CISQ specification features a section defining required attributes and inputs for an implementation to claim compliance with the specification (CISQ, 2012, p. 11):

Automated The CISQ-specification states: "... the analysis of the source code and the actual counting must be fully automated". This means that an implementation cannot only provide an interface where statistics are shown, but an automatic analysis based on the data must also be included.

Objective The analysis must be repeatable: "Two independent analyses of the same application must produce the same counts for each of the Quality Measure Elements measured as part of a Software Quality Characteristic".

Transparent All required inputs and all generated outputs must be clearly listed to the user. Object Management Group suggests: "Implementations of this specification are encouraged to provide a list of inputs and outputs at interim stages of the analysis process".

Verifiable An implementation must "...state the assumptions/heuristics it uses with sufficient detail so that the calculations may be independently verified by third parties.". Inputs that are used must also be "...clearly described and itemized so that the can be audited by a third party". A document with assumptions and the inputs used must therefore also be provided.

Required inputs To claim compliance, an implementation must also require the following inputs:

1. The entire source code for the application being analyzed
2. Documentation or information about how the layers of tiers in the design of the application [are structured] and how functions are allocated to them
3. Information about whether the design is based on DOM or SAX in order to evaluate multiple inheritance mechanisms
4. A list of vetted[3] libraries [that] are being used to "neutralize" input data
5. What routines / API calls are being used for remote authentication, to any custom initialization and cleanup routines, to synchronize resources, or to neutralize accepted file types or the names of resources
6. The encryption algorithms that are being used

2.4.7.4 Software Quality Calculation

The specification proposes a formula for calculating the score for each quality characteristic. The mathematical definition provided is (CISQ, 2012, p. 28):

Definition 2.5 ***Quality Characteristic Score***

$$QC_j = \sum_{k=1}^{n} (Quality\ Measure\ Element_{jk})$$

where $_j$ is a quality characteristic and $_{jk}$ is a quality measure element within that characteristic.

Described in basic terms, the score for one quality characteristic is the sum of rule violations found for each of its quality measure elements (see Table 2.3). For example, the quality score for the *maintainability* of a program is the sum of all violations in the program which are found by the maintainability measures.

[3] Trusted and verified by a larger community

TABLE 2.5: CISQ Performance Efficiency Measures implemented by Satrijandi and Widyani (2015).

CISQ Measure	Description
P02	# of SELECTS done through sequential searches
P03	# of complex queries on very large tables, where complex query $=\geq 5$ joins, sub-queries ≥ 3
P04	# of indices in very large tables > 3
P05	# of very large tables with $^{3}3$ indices
P06	ratio of # child objects of DOM compared to child objects of SAX (DOM / SAX < 1.0)
P07	# of loops with expensive operations
P08	# of initializations inside static blocks
P11	# of unmatched allocation/de-allocations of memory for objects
P12	# of additional immutable objects
P13	# of object references that lack a destructor/finalize function
P15	# of static variables/collections/objects that are not singletons

2.4.8 Efficiency Measurement of Java Android Code

This study, by Satrijandi and Widyani (2015), proposes the use of the CISQ Specifications for Automated Quality Characteristic Measures for evaluating efficiency of Java Android code. The static analysis tool PMD was used to analyze the existing application Daily Money with the added measures implemented from the CISQ specification (Satrijandi and Widyani, 2015, p. 1).

2.4.8.1 Measures

While the CISQ specification includes 16 measures for performance efficiency, 11 of them were implemented in the project. The remaining five are measures 1, 9, 10, 14 and 16, and were excluded because the researchers deemed them as not applicable to Java Android code (Satrijandi and Widyani, 2015, p. 2). The implemented measures are listed in Table 2.5.

2.4.9 Tools

Satrijandi and Widyani (2015) list four static code analysis tools:

- Checkstyle
- Findbugs

- PMD
- Android Lint

PMD was chosen due to its analysis approach using lexical analysis or AST, extendability, documentation, active development, and community (Satrijandi and Widyani, 2015).

2.4.9.1 Evaluation Methods

Three evaluation methods were used: unit tests, stress tests and benchmarking tests. The unit tests were created to make sure the rules were implemented correctly to specification, while the stress tests were run to verify that the measures still worked properly on large scale test cases. The benchmarking tests displayed the speed of the measures (Satrijandi and Widyani, 2015).

2.4.9.2 Conclusions

The researchers conclude that the CISQ performance efficiency measures were successfully implemented for analyzing Java Android code, and that static code analysis can reduce the work effort during testing phases of developing Java Android applications (Satrijandi and Widyani, 2015, p. 5).

2.4.10 On the Validity of the IT-CISQ Quality Model for Automatic Measurement of Maintainability

Plösch, Schürz, and Körner have created an implementation of the Quality Characteristic *maintainability* called *MUSE Understand Scripting Engine* (MUSE) as a tool to study the validity of the CISQ Specifications for Automated Quality Characteristic Measures. The study focuses on maintainability due to importance of the characteristic, and because the authors had access to data gathered from earlier assessment which could be used to validate the standard (Plösch, Schürz, and Körner, 2015, p. 326).

TABLE 2.6: CISQ maintainability measures deemed as unclear by Plösch, Schürz, and Körner (2015).

M1, M2, M3	We assume that every function or method is always part of a layer. The implementation of a function or method is allowed to call functions or methods from direct upper or lower layers, but not from layers that are far away. In order to automatically calculate this measure, it has to be assured (by configuration) that each function or method (or its more high-level constructs like classes or packages) is assigned to exactly one layer.
M8	No meaningful implementation possible for Java. For this reason M8 is not considered anymore.
M11	The fan-out is calculated by summing up the number of called functions (methods) and the number of member variables set.
M12	For the calculation of the object coupling we rely on the well-known metric Coupling Between Object (CBO) as specified in [13]. M18 As threshold value for the cyclomatic complexity, we assume a value of 10 as proposed by McCabe [14] and because of the widespread use of this threshold value.
M19	We allow explicit specification for each project which packages or classes contain data or file operations.

2.4.10.1 Unclear Measures

The CISQ maintainability measures were "... not exact enough..." (Plösch, Schürz, and Körner, 2015), so the authors have made assumptions about the measures which were found to be unclear (M1, M2, M3, M8, M11, M12, M18 and M19), explained in Table 2.6.

2.4.10.2 Tools

MUSE uses the commercial code analyzer *Understand* to extract information about the code. MUSE analyzes the data from Understand and produces an XML file with the results, using one Perl module for each measure to detect the issues. Fig. 2.12 illustrates the architecture of the tool (Plösch, Schürz, and Körner, 2015).

2.4.10.3 Evaluation of Applicability of the CISQ Specification

By implementing the CISQ specification as the MUSE software, Plösch, Schürz, and Körner evaluate the applicability of the specification through comparison with their own quality model, *Expert Centered Method for Internal Software Quality* (EMISQ). The comparison between the models used a benchmarking-based approach, where five open-source projects were analyzed by MUSE and compared against an existing base

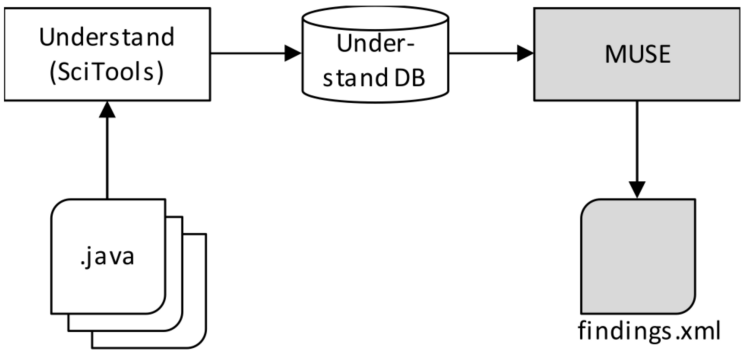

FIGURE 2.12: Tool architecture for implementing IT-CISQ measures (Plösch, Schürz, and Körner, 2015)

of other open-source projects. The following five open-source projects were evaluated with the software (Plösch, Schürz, and Körner, 2015, p. 330):

- Log4j version 1.2.15
- JabRef version 2.3.1
- TVBrowser version 2.2.5
- RSSOwl version 1.2.4
- Checkstyle version 4.4

The data used for benchmarking was the count of rule violations for each maintainability measure per project, normalized by a set of provided size metrics. The same approach was made with the EMISQ model, and a comparison of the resulting ranks of the open-source projects was made. Both models concluded that *Checkstyle* was the best project, ranking as #1, and *JabRef* was the worst, ranking at #5, in terms of software quality.

2.4.10.4 Conclusions

Plösch, Schürz, and Körner assess the CISQ standard to be implementable. The results from their tool correlate with results from previous research with manual expert evaluations. However, the authors claim that the CISQ maintainability measurements cover less aspects of maintainability than the EMISQ model, and that they are "... only partially suitable for improvement programs..." (Plösch, Schürz, and Körner, 2015, p. 334).

2.5 Chapter Summary

This chapter has presented theory and related research within software engineering and software quality analysis. Software engineering comprises structured techniques for specifying, designing and developing software. Analyzing the quality, and specifically the maintainability of software is possible with existing models, standards and metrics which can be calculated automatically through defined algorithms. Two partial implementations of the *CISQ Specifications for Automated Quality Characteristic Measures* have been made. The configured version of PMD by Satrijandi and Widyani (2015) implements the *performance efficiency* characteristic for Java Android-based software, and the MUSE software by Plösch, Schürz, and Körner (2015) implements the *maintainability* characteristic for Java-based software.

Chapter 3

Research Method

In this chapter, the research and development methods practiced in this research project are presented and explained. First, the Design Science Research (DSR) framework is presented, where the seven guidelines of DSR by Hevner et al. (2004) is detailed. After this, the research evaluation method is described, and lastly, the research design is presented.

3.1 Design Science Research

DSR is a research methodology framework for researching Information Systems (IS) and Software Engineering, where research questions are answered by modeling and implementing a solution. In addition to solving the research questions, such a solution can be used as a more practical guideline or example for the commercial industry, which in turn can further develop the solution for a larger scope (Hevner et al., 2004) (see Fig. 3.1). Hevner et al. describe seven guidelines for Design-Science Research, as shown in Table 3.1 and described in further detail in the subsections below.

3.1.1 Guideline 1: Design as an Artifact

It is important that a DSR-project leads to "a purposeful IT artifact" (Hevner et al., 2004, p. 82); a proof-of-concept which is described to such an extent that feasibility of the design process and of the designed product is demonstrated.

3.1.2 Guideline 2: Problem Relevance

A DSR-project must be relevant to the problem space, in the sense that the research must relate to problems and opportunities in the intended domain (Hevner et al.,

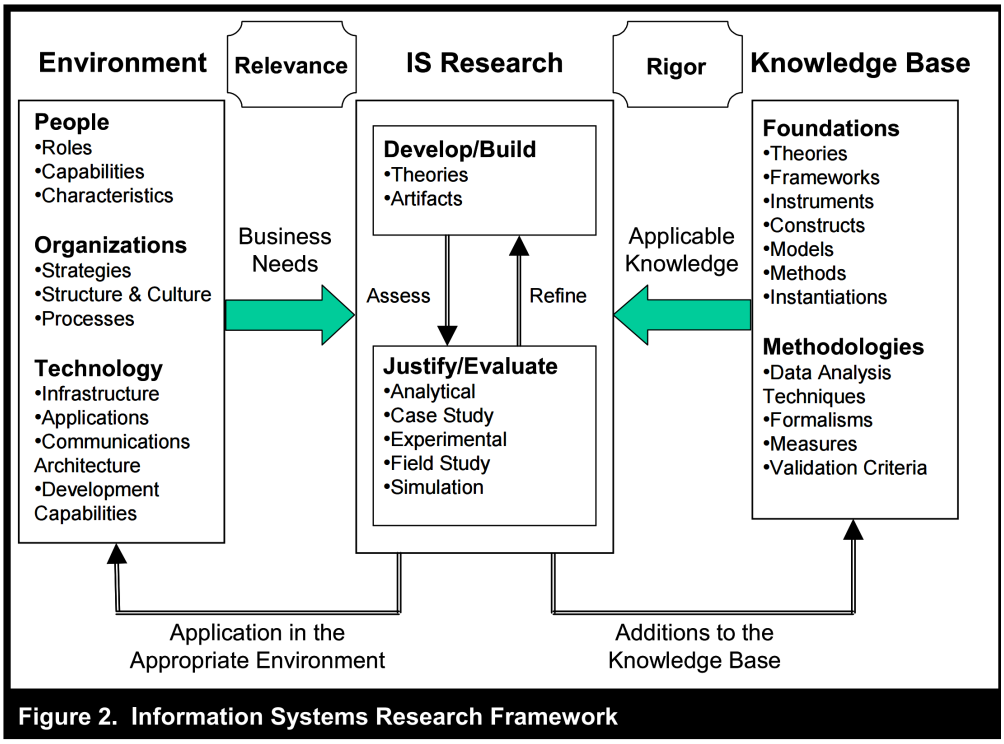

FIGURE 3.1: Information Systems Research Framework (Hevner et al., 2004, p. 80).

TABLE 3.1: Design-Science Research Guidelines (Hevner et al., 2004, p. 83).

Guideline	Description
1: Design as an Artifact	Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.
2: Problem Relevance	The objective of design-science research is to develop technology-based solutions to important and relevant business problems.
3: Design Evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.
4: Research Contributions	Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.
5: Research Rigor	Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.
6: Design as a Search Process	The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.
7: Communication of Research	Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.

TABLE 3.2: DSR Design Evaluation Methods (Hevner et al., 2004, p. 86).

Evaluation type	Subtypes
1. Observational	Case Study: Study artifact in depth in business environment
	Field Study: Monitor use of artifact in multiple projects
2. Analytical	Static Analysis: Examine structure of artifact for static qualities (e.g., complexity)
	Architecture Analysis: Study fit of artifact into technical IS architecture
	Optimization: Demonstrate inherent optimal properties of artifact or provide optimality bounds on artifact behavior
	Dynamic Analysis: Study artifact in use for dynamic qualities (e.g., performance)
3. Experimental	Controlled Experiment: Study artifact in controlled environment for qualities (e.g., usability)
	Simulation: Execute artifact with artificial data
4. Testing	Functional (Black Box) Testing: Execute artifact interfaces to discover failures and identify defects
	Structural (White Box) Testing: Perform coverage testing of some metric (e.g., execution paths) in the artifact implementation
5. Descriptive	Informed Argument: Use information from the knowledge base (e.g., relevant research) to build a convincing argument for the artifact's utility
	Scenarios: Construct detailed scenarios around the artifact to demonstrate its utility

2004, p. 85). In other words, in IS research, a DSR-project must facilitate further advancements for the community which facilitated the project, so that further development and research can be performed.

3.1.3 Guideline 3: Design Evaluation

The DSR artifact must be thoroughly evaluated. These methods require a set of evaluation metrics - attributes of the artifact which can be measured and are appropriate to the problem space. It is also crucial that the evaluation methods that are selected match the designed artifact and the selected evaluation metrics (Hevner et al., 2004, p. 85). In Table 3.2, the five types (and subtypes) of design evaluation methods Hevner et al. (2004) are described.

3.1.4 Guideline 4: Research Contributions

Not only must the project provide knowledge and insight to the community which facilitated the project, but the project must contribute to the domain of the design

artifact, knowledge of the design construction, or methodologies for evaluation. Assessment of contribution relies on criteria focusing on *representational fidelity and implementability*. This means the artifact must clearly represent the environment of business and technology used in the research, and be implementable in the real world. In addition, the project must also be a well-contributing solution to an important problem in the commercial environment (Hevner et al., 2004, p. 87).

3.1.5 Guideline 5: Research Rigor

A DSR-project must be well-balanced between rigor and relevance, in that too rigorous methods for creation and evaluation can result in low relevance, and that too imprecise or flexible methods can result in low validity of the data, which indirectly also lowers relevance. (Hevner et al., 2004).

3.1.6 Guideline 6: Design as a Search Process

A DSR-project is an iterative search for a satisfactory solution to a problem. It is not required, and arguably not possible, to specify all the possible design solutions to an IS-problem, but the continuous, iterative create/evaluate cycle (or Generate/Test Cycle) provides an organic way of finding a solution which satisfies as an answer to the problem, and confines itself to the boundaries of the identified laws of the system. It is not the main goal to figure out why the artifact works, but that it works, and how (Hevner et al., 2004).

3.1.7 Guideline 7: Communication of Research

Since a DSR-project is required to contribute to both technology-oriented and management-oriented audiences, tailoring the communication of the research is particularly important. While technology-oriented audiences are looking for precise details for implementing the artifact in an organization, management-oriented audiences need a different set of details to decide whether or not to commit to constructing or purchasing the artifact for their organization. Therefore, with a set of two very different main audiences, the project must be described in such a way that it provides the necessary information for both implementation and integration (Hevner et al., 2004, p. 90).

3.2 Research Design

This section explains how the research questions (see Section 1.3) will be answered using the DSR framework, development methods, and evaluation and validation techniques, resulting in the design, implementation and evaluation of the open-source software SQUIDS. First, a repetition of the research questions:

Q1 - How can a static code analyzer be developed as an Eclipse plug-in which, based on a standard, detects maintainability problems in a software project?

Q2 - How can such an implementation visualize maintainability problems in a software project to the user?

Q3 - Can such an implementation give a performance which makes it usable during development?

3.2.1 Using DSR

Since this research project focuses on the design and implementation of an IT solution to a problem, the Design Science Research framework (see Section 3.1) is chosen as the research method. In order to answer the research questions (see Section 1.3), the project will follow the seven DSR guidelines from Hevner et al. (2004) (see Table 3.1). Application of the DSR guidelines in the project is described in Table 3.3.

3.2.2 Development Method

Answering Q1 (see Section 1.3) and developing the artifact requires a structured method for software development. In order to manage the development of the software, a development method with a set of processes and techniques will be used. These are primarily collected from the agile method eXtreme Programming (XP) (see Section 2.1.7), although some modifications are made to better suit a development team of one.

3.2.2.1 User Stories

Requirements for SQUIDS will be written as *user stories* (see Section 2.1.3), using the usual pattern: *As a <role>, I want <goal/desire> so that <benefit>*. The roles will be either *developer* who is using the software, or *future collaborator* engaged in further

TABLE 3.3: Application of the seven DSR guidelines.

Guideline	Application
1: Design as an Artifact	An open-source software named SQuIDS (**S**oftware **Qu**ality **I**ssue **D**etection **S**ystem) will be developed as an *artifact*, implementing the CISQ maintainability characteristic.
2: Problem Relevance	Since a tool for analyzing software maintainability is to be developed to address software quality problems in software development, the research will be relevant to the problem space.
3: Design Evaluation	The artifact will be *evaluated* by combining dynamic analysis, functional testing, and informed arguments from existing research (see TABLE 3.2).
4: Research Contributions	A practical, implementable (installable) tool for software developers will be *contributed*, as well as a description of how it was designed and developed to facilitate further research.
5: Research Rigor	By using methods for design, development, evaluation and validation, the research will be *rigorous*.
6: Design as a Search Process	The software will be developed in iterations, *searching* for answers to the research questions. Q2 is especially relevant to this guideline, as the speed of SQuIDS is important if it is to be used often during a software development project. Therefore, optimization of the performance of the artifact will be iteratively improved throughout development.
7: Communication of Research	This guideline will be followed with a modification - the *communication* of the research must not only be tailored for both technology-oriented and management-oriented audiences, but also for the academic readers in the committee which will evaluate this thesis.

development or using parts of the software for other purposes. This way, both intended users and the relevant people in the software community (see Section 3.1.4) are considered during development.

3.2.2.2 Iterative Development

To ensure being able to improve the design according to DSR guideline 6 (see Section 3.1.6) and quickly respond to change (agility), development of SQuIDS will be divided into iterations. As mentioned in Section 3.2.1, optimization of the performance of the software will be iteratively improved upon, as an increasing amount of the CISQ maintainability measures are implemented. In order to keep user stories small enough for one developer, and keep progress visible, iterations will be set to one week in this project. Iteration planning will be conducted at the start of each iteration. In the project, user stories will only be broken down into tasks when

needed. This is to avoid spending too much time on planning, as the project has only one developer.

3.2.2.3 Estimation and Project Velocity

User stories will be estimated, and project velocity measured during development to assess how much work can be completed, and at the end of the project, how much work is needed for creating the artifact. This will answer a part of Q1, where a measure of the workload for the development of a software is part of determining how it may be implemented.

In contrast to what is proposed for XP, user stories will not be estimated in the linear 1, 2 or 3 week scale (Wells, 1999c). Instead, a more fine-grained, progressive scale inspired by Cohn (2006) will be used:

- **1 story point**: 1-4 hours
- **2 story points**: ½-1 day
- **4 story points**: 1-2 days
- **8 story points**: 2-5 days
- **16 story points**: 1-2 weeks

This scale allows for smaller user stories and visualizing the uncertainty of how much work the larger user stories are, by doubling the points for each step. The story points will not only function as an indicator of how much work is remaining or completed. It will also be used for calculating the project velocity.

3.2.3 Research Evaluation

As stated in Section 3.2.1, evaluation of the artifact SQuIDS will be a combination of dynamic analysis, functional testing and informed arguments. The performance of SQuIDS will be measured by letting it analyze existing software projects and measuring the time it uses compared to the project sizes, giving a dynamic analysis of the artifact. The implemented measures will be validated with Black Box unit tests, thus performing a functional test. Compliance with the required attributes and inputs in the CISQ specification, and comparison of results between related work and SQuIDS will be providing an informed argument (descriptive) of the artifact's utility and correctness.

3.2.3.1 CISQ Specification Compliance

As a part of answering Q1, compliance to the CISQ specification will be evaluated according to how well the CISQ maintainability measures which are implemented, and the requirements listed in the specification (see Section 2.4.7.3). However, irrelevant requirements for a partial implementation (only maintainability measures) will be ignored. Additionally, CISQ maintainability measure M08: *# of instances of multiple inheritance of concrete implementation classes (threshold* > 1*)* does not apply to Java. In the Java programming language, a class cannot extend multiple implementation classes, which makes this CISQ measure obsolete (Oracle, 2015b).

3.2.3.2 Unit Test Validation

Black Box unit tests (see Section 2.1.8) will be written for the CISQ measures to determine the validity of their implementations. These will test whether or not the implementations of the CISQ measures may fail to meet the definition of the quality measure.

The unit test design for SQUIDS requires each unit tests for implemented CISQ measures to:

1. Be written before the implementation, thus following TFP (see Section 2.1.8)
2. Test the `analyzeNode()` method which is present in every CISQ measure
3. Cover all expected boundary values and types of input which should generate issues
4. Have a line coverage of $\geq 90\%$, ensuring that the most important parts of the measure is testable

Additionally, other possible weaknesses will be handled for individual CISQ measures. This gives the test design a semi-formal approach.

The quality of the tests will themselves be tested with line coverage analysis and mutation testing, using the PITEST plug-in for Eclipse[1]. The PITEST analysis will be limited to the unit tests written for the CISQ measures, using the standard settings of PITEST.

[1]The PITEST plug-in for Eclipse is called Pitclipse: https://github.com/philglover/pitclipse

3.2.3.3 Comparison of Results with Related Work

To further validate the implementation of the CISQ maintainability measures in this research project, the same projects analyzed by the MUSE tool (see Section 2.4.10) will be analyzed with SQUIDS, and the results will be compared with the raw results from MUSE. The raw results were not available in Plösch, Schürz, and Körner (2015), but were provided by Schürz (2016) over e-mail. The software projects and versions are:

- Checkstyle version 4.4 [2]
- JabRef version 2.3.1 [3]
- Log4j version 1.2.15 [4]
- RSSOwl version 1.2.4 [5]
- TVBrowser version 2.2.6 (2.2.5 used by Plösch, Schürz, and Körner (2015) was not available) [6]

The TV-Browser project may be excluded from this comparison, since the version used by Plösch, Schürz, and Körner (2015) is no longer available. First, a simple count of maintainability problems found in the software projects by the two analyzers will be compared per measure, and discussed. Secondly, individual problems reported by a selection of CISQ measures will be manually inspected with the following purposes:

1. Search for erroneously reported problems (false positives) and unreported, real problems (false negatives)
2. Discuss the correctness of MUSE and SQUIDS
3. Provide a means of validating the correctness of SQUIDS

In order to discuss individual findings, results from SQUIDS must be converted to the same XML format as the results from MUSE, and a software must be created to find problems which are only reported by one of the implementations.

3.2.3.4 Visualizing Maintainability Problems to a User

SQUIDS will be implemented with a working example of how maintainability issues can be visualized to a user, to answer Q2. The project will not seek an optimal solution to this problem, but provide a proof-of-concept and discuss its strengths and weaknesses compared to other possible solutions.

[2] https://logging.apache.org/log4j/1.2/source-repository.html commit 3fb7ae8e (Git)
[3] https://logging.apache.org/log4j/1.2/source-repository.html
[4] https://logging.apache.org/log4j/1.2/source-repository.html revision 569611 (SVN)
[5] https://logging.apache.org/log4j/1.2/source-repository.html
[6] https://sourceforge.net/projects/tvbrowser/files/TV-Browser Releases...

3.2.3.5 Measuring Performance

Q3 requires that the performance of the artifact is optimized. Using all the open-source software projects which were analyzed by MUSE (see Section 3.2.3.3), the performance of SQUIDS will be measured according to the time it uses on analyzing the projects, and the project sizes. On the same computer setup (see Table 3.4), for each project, the time used by SQUIDS to analyze the project compared with the number of Java files and the total physical LOC (see Section 2.4.5) in the project will be measured, to provide an indication of SQUIDS's performance. Before evaluation, this analysis will be used to target problematic areas in SQUIDS in order to improve performance.

TABLE 3.4: Setup for performance testing of SQUIDS.

Computer make and model	Lenovo Yoga 2 Pro
Memory	7.7GB RAM
Processor	Intel® Core™ i7-4500U CPU @ 1.80GHz × 4
Operating system	Ubuntu 15.10
Java version	Java SE 1.8 (Oracle Java 8)
Environment	Eclipse 4.5.2 (Mars.2) build ID 20160218-0600 with VM arguments `-Dosgi.requiredJavaVersion=1.8 -Xms40m -Xmx512m`
Active programs and services	• Google Chrome 50.0.2661.75 beta (64-bit) • Document Viewer 3.16.1 • WiFi (Intel® Wireless-N 7260 802.11 b/g/n)

3.3 Chapter Summary

This chapter has presented the Design Science Research (DSR) framework, and how it will be used to answer the research questions. A software named SQUIDS (**S**oftware **Qu**ality **I**ssue **D**etection **S**ystem) will be developed using practices from eXtreme Programming. Correctness of SQUIDS will be tested with unit testing, comparison with MUSE (Plösch, Schürz, and Körner, 2015) and manual inspection of analysis results. Performance will be tested to see if an acceptable speed can be achieved, and a working example of how maintainability issues can be displayed to a user will be implemented.

Chapter 4

Development Process

This chapter describes how the software SQuIDS was designed and developed. It details how requirements were specified and implemented, how the software was modeled, and which technical and theoretical challenges were encountered during the 17 iterations of development.

4.1 User Scenarios

The intended users for SQuIDS are Java developers using the Eclipse IDE. The IDE was chosen due to personal experience. The two user scenarios are explained in the following subsections.

4.1.1 Scenario 1: Evaluate own software

The first scenario, illustrated in Fig. 4.1, is a developer using the software to receive feedback on the code quality of the software under development. In this scenario, the goal of the developer is to identify maintainability problems while programming, to reduce future maintenance effort.

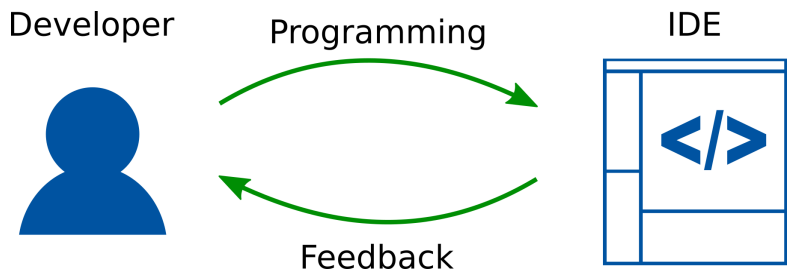

FIGURE 4.1: Developer using the software to improve code quality.

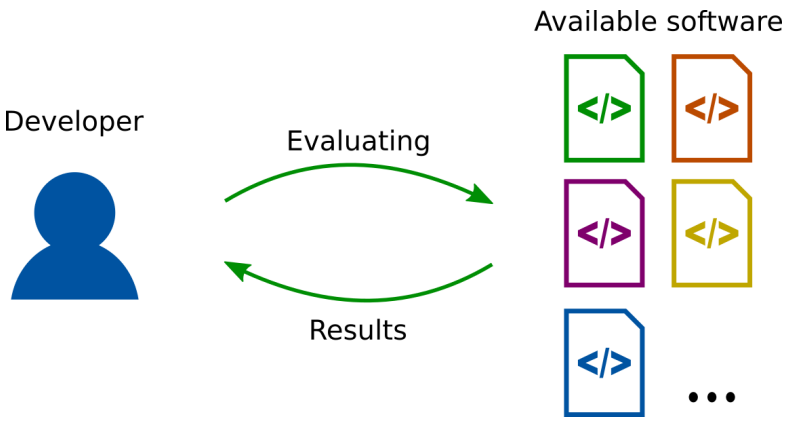

FIGURE 4.2: Developer evaluating a software before purchasing.

4.1.2 Scenario 2: Evaluate external software

The second scenario, illustrated in Fig. 4.2, is a developer using the software to evaluate external software. In this scenario, the goal of the developer is to get a view of how maintainable an available software is, thus helping in decision making when considering external software to use in a development project. Using a more maintainable external software reduces the risk of future maintenance effort (Coleman et al., 1994, p. 46). This may be especially useful when considering unfamiliar, open-source libraries.

4.2 Requirements

Software requirements (see Section 2.1.1) were specified for SQUIDS in order to define a structured scope of functionality. Due to time constraints, the project did not collect requirements from interaction with people, such as user testing or organizational involvement. However, the requirements of SQUIDS were specified for the intended users, based on the user scenarios. In addition, requirements were collected directly from the CISQ compliance-list and the maintainability measures, as implementation of the maintainability characteristic was the main goal of the project.

4.2.1 Functional Requirements

The final functional requirements of SQUIDS are listed below.

1. The software must be able to detect all CISQ maintainability problems according to the CISQ maintainability measures, except M08 (see Section 3.2.3.1)

2. The software must be configurable to specify the architectural layers in the software under analysis
3. The software must be configurable to specify which packages and classes contain database or file operations
4. The software must be able to display the problems to the user with the following information:
   (a) The problem description
   (b) The location in the source code where the problem occurs
5. The software must provide a separate report for each project, including:
   (a) The counts of rule violations for the CISQ maintainability measures
   (b) The QCj score for the maintainability characteristic (see Section 2.4.7.4)

4.2.2 Nonfunctional Requirements

The final nonfunctional requirements of SQuIDS are listed below.

1. The software should be installable as an Eclipse IDE plug-in
2. The software should need as little configuration by the user as possible
3. Problems must be visualized in an manner which makes it easy for the user to identify the section of code to improve.
4. The software should be usable by developers for other purposes, such as developing a plug-in for another IDE
   (a) The software should be released as an open-source project
   (b) The part of the software which deals only with analyzing files should be platform- and framework-independent
5. Analysis of a software project should be performed quickly.
   (a) A project of 10 Java files and a total of 1,000 physical LOC should take less than 1 second to analyze, where 100 Java files for 10,000 physical LOC should take less than 10 seconds (Complexity should be linear or less, i.e. $O(n)$)
   (b) A visual percentage-done indicator should be provided (see Section 2.1.5)

4.2.3 User Stories

For SQuIDS, requirements were written as user stories (see Section 2.1.3) for both the CISQ maintainability measures and the plug-in's functionality required by users. The final list of user stories for the plug-in is displayed in Appendix A, in the order they were implemented. User stories 4-7 and 9-26 are direct adaptations from the CISQ

maintainability measures. For example, the CISQ maintainability measure 2: *# of layer-skipping calls* is adapted into user story 4: *As a developer I want to be notified about layer-skipping calls so that I can improve code architecture*.

4.3 Design

Software design is a creative process where customer requirements are turned into system components and the relationships between them (Sommerville, 2011, p. 194). The design of SQuIDS was developed through iterations, and underwent many changes during development. In the following sections, the final context- and class diagrams for SQuIDS are displayed.

4.3.1 Context Diagram

A context diagram is a high-level model of the interaction between the system and the environment (Vliet, 2007, p. 357). As shown in Fig. 4.3, SQuIDS communicates with the Eclipse IDE alongside other plug-ins. The Eclipse IDE gives access to source code files and the ability to add visible markers in the Eclipse editor.

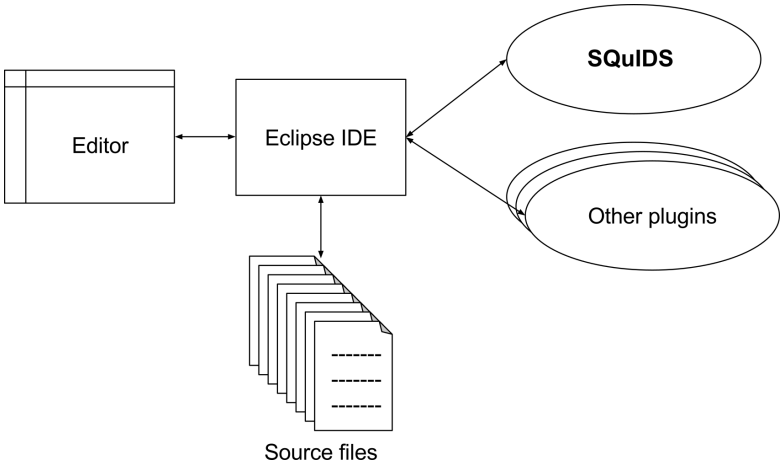

FIGURE 4.3: SQuIDS context diagram.

4.3.2 Class Diagram

Fig. 4.4 displays a UML class diagram (see Section 2.1.4.2) of the relations between the classes in SQuIDS, with fields and private methods hidden. Static methods and

classes are underlined, and abstract methods and classes are italic. A detailed UML class diagram of the inheritance hierarchy of the `Measure` classes is available in Appendix C.

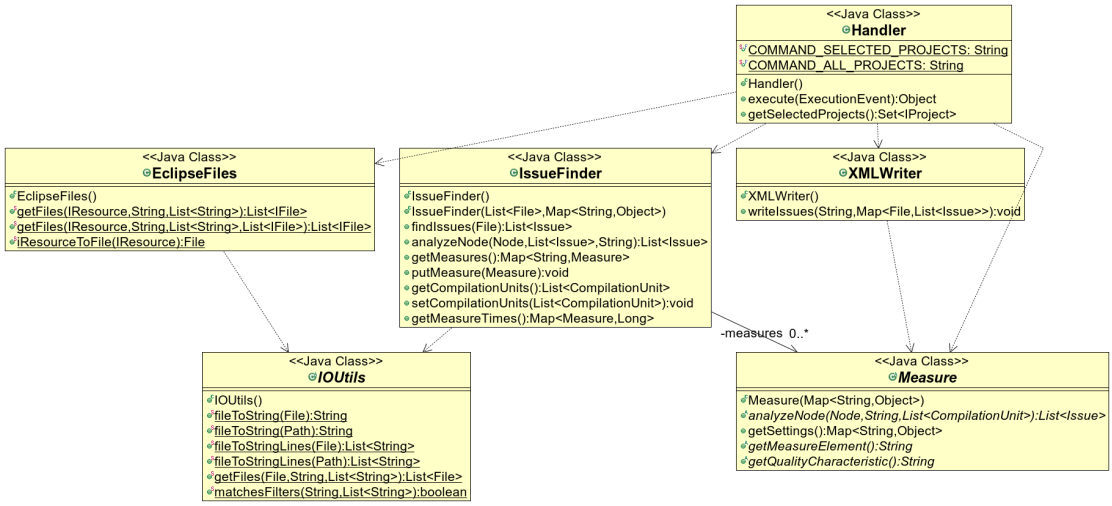

FIGURE 4.4: SQUIDS class diagram.

4.3.3 Process Diagram

The high-level process in SQUIDS is illustrated as a BPMN diagram (see Section 2.1.4.3) in Fig. 4.5. The user requests an analysis of the code, which initiates the plug-in. SQUIDS requests files from the Eclipse IDE, analyzes them, and creates problem markers, which the Eclipse IDE displays in the editor. The subtask of analyzing files is displayed in Fig. 4.6. For each node in each file, all measures are run, and create issues if problems are found, which are added to a list.

4.4 Tools

Developing SQUIDS required a certain set of tools in order to implement the CISQ maintainability measures to a useful tool for developers. This section explains how and why these were used.

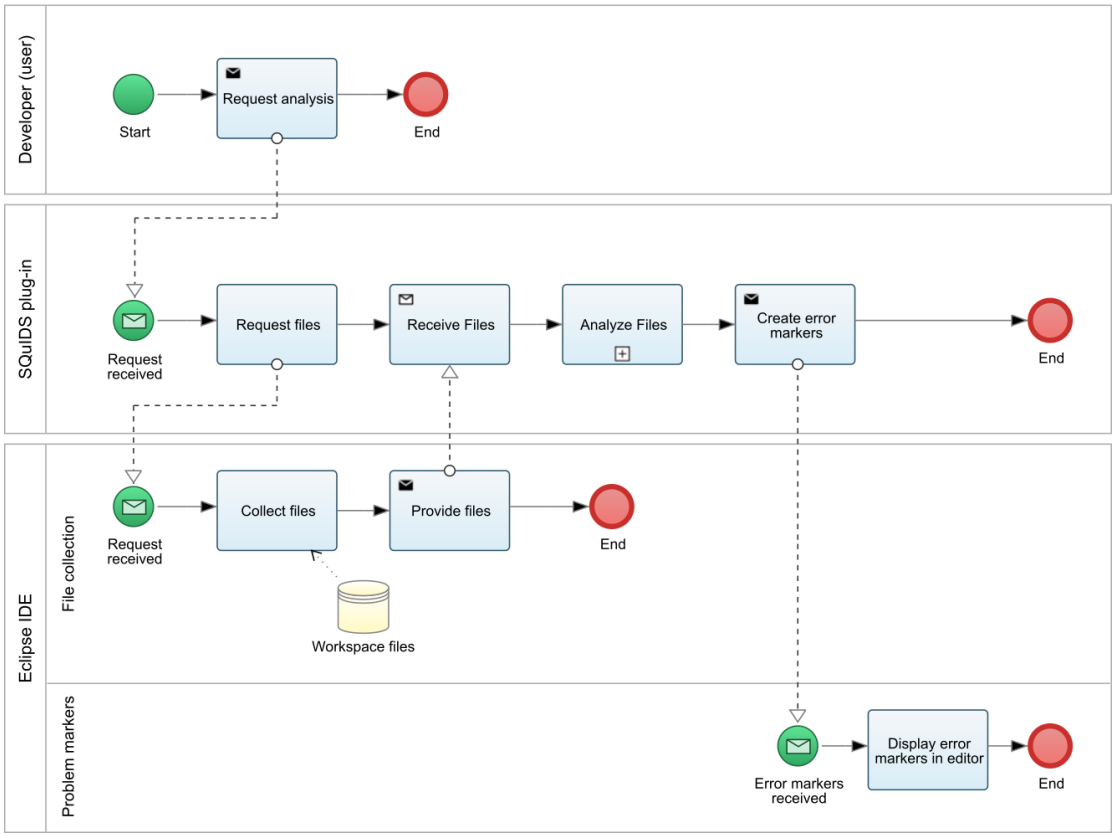

FIGURE 4.5: High-level BPMN process diagram of SQUIDS.

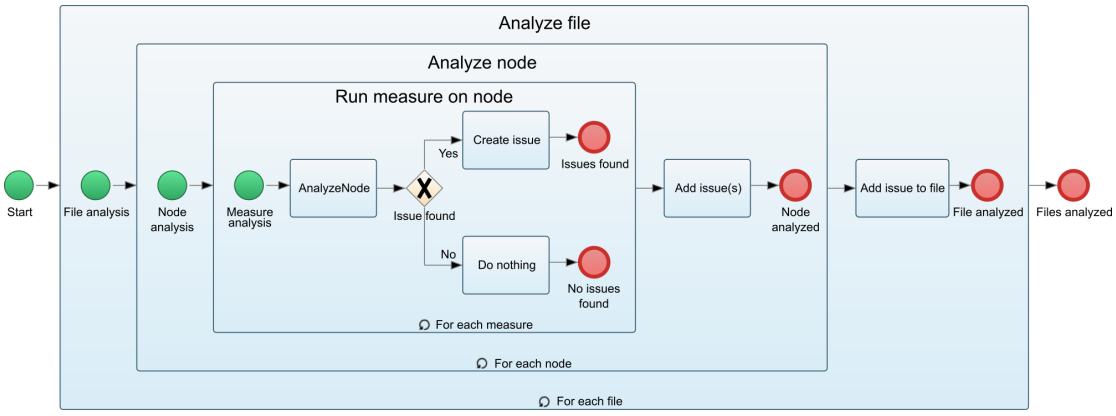

FIGURE 4.6: Low-level BPMN process diagram of file analysis in SQUIDS.

4.4.1 Programming Language

The programming language Java was selected both as the language for implementing the software, and as the language which the software would be able to analyze. In other words, write in Java to analyze Java. The choice of implementation language was made due to both personal expertise in Java, and that the language is platform-independent (Oracle, 2015a), which means that any operating system supporting Java, would also support the software. These reasons also apply to the choice of language to analyze. Additionally, the strong typing in Java (Cornell University, 2005) was an important aspect, as it makes analysis more feasible.

4.4.2 JavaParser

In order to implement the CISQ maintainability measures, a way to analyze code had to be selected.

4.4.2.1 Regex

One option which was considered early in development was using Regular Expressions (regex), a text pattern-matching technology. Its usage is in concept creating a search pattern, following the regex syntax, feeding it text, and receiving a list of matching results. The technology is widespread in programming practice, and is often an effective and easily implementable solution for text-search and -editing (Goyvaerts, 2015).

While regex could have been used to implement many, or perhaps all of the measures, it would be difficult to create a fast and maintainable solution using regex for all the measures. A regex solution would have to search the same text for each measure in order to identify different problems, resulting in a performance-heavy and time consuming analysis. Furthermore, some measures are concerned with the relationships between constructs such as packages, classes and methods. This means either searching through multiple files or searching the same file more than once per measure, or both.

4.4.2.2 Abstract Syntax Tree

Instead of searching for patterns in text, creating and traversing Abstract Syntax Trees (AST) (see Section 2.4.1) of the Java files proved to be a strategy which could handle

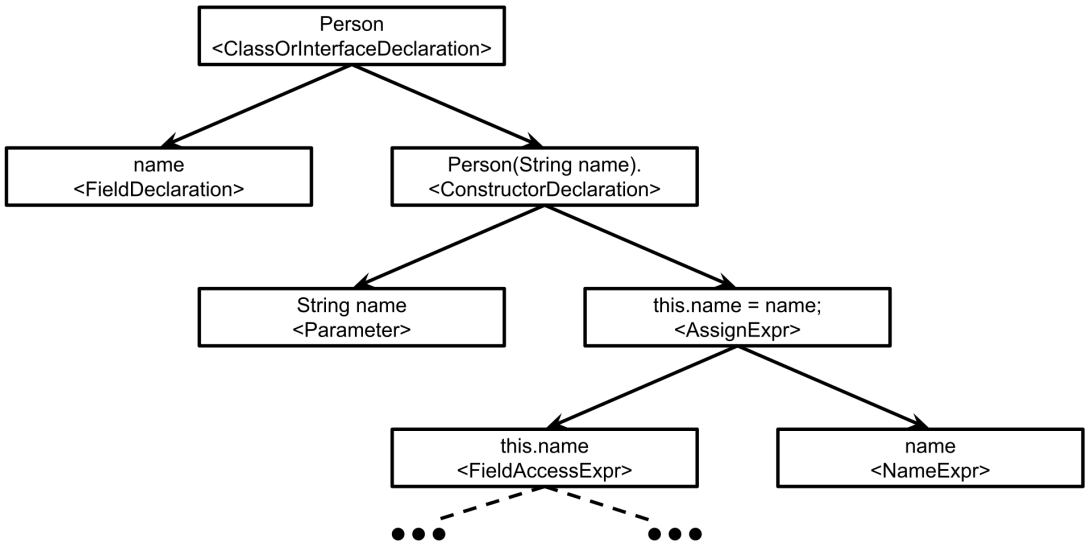

FIGURE 4.7: Example excerpt from a JavaParser-produced AST.

complex queries on code. Several open-source tools for generating AST from Java files exist, such as:

- JavaParser
- Eclipse JDT
- javalang
- jLaTo

The JavaParser library[1] was selected over the other alternatives for having both available documentation (in Javadoc) and a richer Application Programming Interface (API) than the others. An example of how the AST produced by JavaParser is structured, is illustrated in Fig. 4.7.

4.4.3 Eclipse PDE

The Eclipse Plug-in Development Environment (PDE) is an addition to the Eclipse IDE which extends it with a tool suite for creating, running, testing and deploying plug-ins for Eclipse. This extension, along with the Eclipse IDE as platform, were chosen as the development environment, being the only combination found suitable for developing Eclipse plug-ins.

[1] JavaParser version 2.3.0 was used.

4.4.4 Other tools

Additionally, the following tools were used in the project:

- Git - A version control system (VCS) used for systematically saving the project online.
- SmartGitHg - A Graphical User Interface (GUI) client for visually managing Git
- Bitbucket - A web-based hosting service for Git (or Mercurial, another VCS)
- Trello - A web-based project management application. Used in the project as a virtual whiteboard, to manage iterations and user story progress
- Apache Maven - A build automation tool. Used in the project for handling dependencies to for example the JavaParser library.

4.5 Technical Challenges

The project encountered some challenges with the CISQ specification and the JavaParser library during development, which is important to note for future research involving either resources.

4.5.1 Ambiguous CISQ Terms

Some of the terms used in the CISQ maintainability measures are ambiguous, and definitions are neither included within the document, nor referenced externally. Some measures therefore required further research. Plösch, Schürz, and Körner (2015) touch upon most of the same issues in their article On the Validity of the IT-CISQ Quality Model for Automatic Measurement of Maintainability. Their assumptions and conclusions proved valuable in specifying the terms.

4.5.1.1 Layer

The first three CISQ maintainability measures (M1, M2 and M3) (see Table 2.4), which all concern architectural layers in software (see Section 2.4.2). With no provided definition of *layers* in the CISQ specification, it was difficult to implement the measures as intended by the specification.

In SQuIDS, *layers* are considered as defined by MSDN (2009), Xu and Wan (2015) and Mitra (2008), and subscribe to the assumptions of Plösch, Schürz, and Körner (2015) about how functions or methods are assigned to layers, and how they should behave with functions or methods from other layers. Differences between the terms *layer* and *tier* are not taken into account, as their distinctions are inconclusive, and instead referred to both as *layer*.

Mitra's definition of horizontal layers complies with the rest of the definitions of *layers* and the horizontal layers from the CISQ maintainability rule 2. For the span of this thesis, MSDN's mention of sub layers and Mitra's definition of vertical layers are ignored for simplicity.

4.5.1.2 Token

Measure 4: # files that contain 100+ consecutive duplicate tokens, is a measure of the amount of code duplication in a project (CISQ, 2012, p. 26). The term *token* was interpreted as Java keywords, identifiers, separators, operators and literals. In other words, comments and whitespace is not considered (see Section 2.4.1).

4.5.1.3 Object Coupling

Measure 12: # of objects with coupling > 7, did not specify which method for evaluating object coupling (see Section 2.4.3) should be used. Like Plösch, Schürz, and Körner (2015), the metric *Coupling Between Object* from Chidamber and Kemerer (1994) was implemented.

4.5.1.4 Method vs. Function

Measures 1, 5, 9, 11, 14, 18, 19 and 20 use the terms method or function, with no indication of a difference between the two subjects. After an email correspondence with CISQ (Douziech, 2016), it was clarified that the terms were ambiguous, and that they were synonymous in the context of the specification. Earlier implementations of the associated measures discriminated between the two terms, and therefore had to be altered.

4.5.1.5 Commented Out Instruction

Measure 14: # of functions with $> 2\%$ *commented out instructions*, uses the term *instruction*, which is ambiguous without further specification. In relation to software source code, it could be any of the following:

- an instructional comment to code maintainers
- a single line of code (see Section 4.5.1.6)
- a computer instruction (see Section 2.4.4)
- a reference to Java Virtual Machine (JVM) instructions (Oracle, 2016).

For simplicity, this measure was implemented to detect AST nodes which are of the type `Expression` from the JavaParser library. `Expression` nodes are pieces of code which accesses, modifies or compares variables or values. This is not the same as JVM instructions, but the list of JVM instructions shares many related items with the list of `Expression` subclasses.

4.5.1.6 Lines of Code

The term LOC (see Section 2.4.5) is used in *Measure 15: # files* > 1000 *LOC*. Due to time constraints, and for simplicity, the measure was implemented to simply count the PLOC, i.e. all the non-empty lines which are not only comments. In other words, empty lines and comments are ignored.

4.5.1.7 Cyclomatic Complexity

Following Plösch, Schürz, and Körner (2015), SQUIDS uses McCabe, Wallace, and Watson's Cyclomatic Complexity metric (see Section 2.4.6) for *Measure 18: # functions with cyclomatic complexity* $\geq$ *a language specific threshold (table to be inserted)*. The threshold value was set to 10, as in Plösch, Schürz, and Körner (2015).

4.5.1.8 Data or File Operation

Measure 19: # of methods with $\geq$ *7 data or file operations* concerns methods with operations on databases or files. In order to support analysis of software using external libraries with data or file operations, simply looking for the built-in Java-methods with such operations is insufficient. Plösch, Schürz, and Körner (2015) present a solution where classes and packages containing such operations can be

specified. By creating a Graphical User Interface (GUI) for settings, the solution was adapted in this implementation.

4.5.2 JavaParser Weaknesses

The JavaParser library proved to have some weaknesses in documentation and functionality, which increased the time spent on using it.

The available javadoc lacked descriptions of almost all classes and methods. Understanding expected behavior of classes and methods was in some cases difficult, and manual inspection of its source code was therefore necessary to ensure correct behavior.

The AST generated by JavaParser (see Fig. 4.7) is created file-by-file, and therefore contains no direct references between classes. References between the use of a class or variable and the declaration of the class or variable were also not available. A lot of time was therefore spent on creating the helper class `JavaParserHelper` with assisting functions needed to analyze source code.

The original location of tree nodes in the source code is stored as a matrix of line- and column numbers, which is unfortunate for an Eclipse plug-in. In Eclipse, markers are created with start- and end-indexes of the source code, as if each file was one long line. Additionally, tab-characters are treated by JavaParser as exactly eight spaces. A custom function for converting lines and columns to indexes, `columnsToIndexes()` was required to correctly mark the location of the problem found visually in the Eclipse editor. This function directly translates tab-characters to eight spaces, which is not only prone to change if JavaParser changes, but will also mark problems in wrong locations if the source files do not use tab-characters for indentation.

4.6 Iterations

The duration of each iteration was set to one week. The first iteration was week 43, 2015, and the last iteration was week 6, 2016. The project had three phases of development: *Creating a Proof of Concept*, *Creating system architecture* and *Implementing measures*. The following sections describe the iterations grouped by these phases.

4.6.1 Iteration 1

The goal of the first iteration was to create a *Proof of Concept* (PoC) of the ability to find a CISQ Quality Issue (see Table 2.3) and display a warning in the Eclipse editor. Implementing a maintainability measure and performing it with the JavaParser were not the focus of the PoC. Therefore, the simple CISQ Reliability measure 1 "# of exception handling blocks such as Catch and Finally blocks that are empty" was implemented using regex (see Section 4.4.2.1).

The sole user story (see Section 4.2.3) for the first iteration was defined as: "As a developer I want to be notified about empty catch-blocks so that I can improve exception handling" (see Appendix A for all user stories), and had the following sub-tasks:

- Plug-in shell
- Find error
- Report error
- Place Marker on error (line)
- Create and run tests

The PoC was successful, which meant that collecting Java files in a project from an Eclipse plug-in, finding a specific pattern and displaying it as a warning in the editor was perfectly possible. The result is shown in Fig. 4.8.

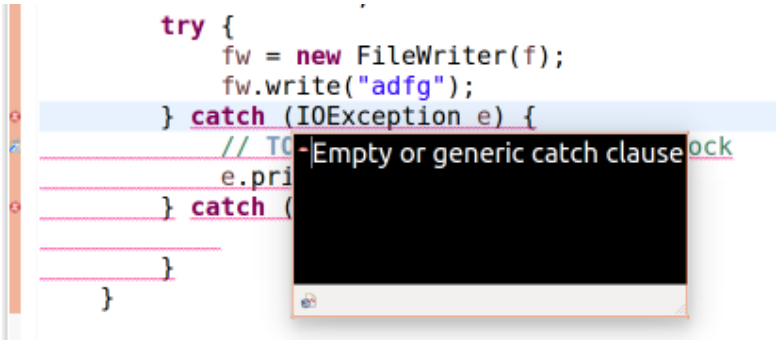

FIGURE 4.8: CISQ Quality Issue warning Proof of Concept.

4.6.2 Iterations 2-11

Weeks 44 through 53, 2015 comprised the second phase of development. Having created a PoC (see Section 4.6.1), the next step was to develop a more maintainable and efficient system architecture for implementing measures. Additionally, more

measures had to be implemented in order to inspect how multiple measures behaved in the system. This was expressed in user stories 2 through 7 (see Appendix A).

The classes `Handler`, `IssueFinder`, `JavaParserHelper` (see Section 4.5.2) and `Measure` were the most important components in the system. Additionally, an abstract unit test class `MeasureTest` was created to facilitate standardized unit testing of measures.

The `Measure` class (see Snippet 4.1) most prominently requires each subclass to call its constructor which accepts a `Map<String, Object>` object for storing settings, and to implement the `analyzeNode()` method which is called recursively for each `Node` in the AST by `IssueFinder`. `analyzeNode()` requires the `Node` being analyzed, a string-representation of the file being analyzed and the list of all Java files (`CompilationUnit`-objects) in the project. The file string is required in order to correctly calculate the location of the `Node` when marking a found problem (see Section 4.5.2). The list of all the Java files is necessary for CISQ measures which rely on relationships between classes[2].

By the end of the iterations in this phase, it also became apparent that the `analyzeNode()` method must be able to return multiple problems in case a CISQ measure finds multiple problems with a single `Node`.

Snippet 4.1: The `Measure` class

```
1  /**
2   * The {@link Measure} class represents each CISQ Automated Quality
3   * Characteristic Measures. It features the simple method
4   * {@link #analyzeNode(Node, String)} which is called for every
5   * {@link com.github.javaparser.ast.Node Node} in the project's AST.
6   *
7   * @author Lars A. V. Cabrera
8   */
9  public abstract class Measure {
10
11     private Map<String, Object> settings;
12
13     /**
14      * Creates a new {@link MeasureTest} with settings.
15      *
16      * @param settings
17      *             - an optional map of settings which might be required for
18      *             certain measures.
19      */
20     public Measure(Map<String, Object> settings) {
21         if (settings == null) {
22             settings = new HashMap<>();
23         }
24         this.settings = settings;
25     }
26
```

[2]CISQ measures M04, M05, M06, M07, M09, M12 and M13 all concern relations between classes

```
27      /**
28       * Analyzes a {@link Node} and the original file string (if required)
29       * according to a specific measure, and returns a list of issues. The list
30       * can contain 0, 1 or > 1 elements. This method is called for each node
31       * in the entire AST, so analysis efficiency is important.
32       *
33       * @param node
34       *             - the Node to be analyzed
35       * @param fileString
36       *             - the original source file string
37       * @param compilationUnits
38       *             - a list of all the CompilationUnit objects in the current
39       *             project.
40       * @return a List of Issue objects, containing none, one or many
         element(s),
41       *         but cannot be null.
42       */
43      public abstract List<Issue> analyzeNode(Node node, String fileString,
         List<CompilationUnit> compilationUnits);
44
45      /**
46       * Returns the settings map which was provided when initializing this
47       * measure.
48       *
49       * @return the settings map which was provided when initializing this
50       *         measure
51       */
52      public Map<String, Object> getSettings() {
53          return this.settings;
54      }
55
56      /**
57       * Returns a description of the type of issue this measure can find, i.e.
58       * the quality measure implemented by the measure.
59       *
60       * @return the type of issue this measure can find
61       */
62      public abstract String getMeasureElement();
63
64      /**
65       * Returns the name of the quality characteristic which the measure
         belongs
66       * to.
67       */
68      public abstract String getQualityCharacteristic();
69
70  }
```

Unit tests were standardized by creating the `MeasureTest` class, which all unit test classes had to extend. `CISQMM06ClassInheritanceLevelTest` (see Snippet 4.2) is the unit test class for CISQ measure M06 (`CISQMM06ClassInheritanceLevel`), and gives an example of the way boundary values are tested in SQUIDS. `MeasureTest` provides two standard methods for testing if a measure finds problems with a specific `Node` or not. Both methods subjects the `Measure` to input, where `findIssue()` asserts that an issue was found when analyzing the node, and `skipIssue()` asserts that an issue was

not found.

Snippet 4.2: Example unit test file for CISQ maintainability measure 6 (shortened for readability)

```
1  /**
2   * Unit test class for {@link CISQMM06ClassInheritanceLevel}
3   *
4   * @author Lars A. V. Cabrera
5   */
6  public class CISQMM06ClassInheritanceLevelTest extends MeasureTest {
7
8      private CompilationUnit classIL6;
9      // ...
10
11     @Before
12     public void setUp() throws Exception {
13         this.issueFinder.getMeasures().clear();
14         this.issueFinder.putMeasure(new CISQMM06ClassInheritanceLevel(new
       HashMap<>()));
15
16         // Load test files
17         File class1 = new File("res/test/inheritance/levels/Class1.java");
18         // ...
19
20         // Parse test files
21         CompilationUnit class1CU = JavaParser.parse(class1);
22         // ...
23
24         // Add parsed test files to list
25         List<CompilationUnit> compilationUnits = new ArrayList<>();
26         compilationUnits.add(class1CU);
27         // ...
28         this.issueFinder.setCompilationUnits(compilationUnits);
29
30         this.classIL6 = class7CU; // Inheritance level: 6
31         this.classIL7 = class8CU; // Inheritance level: 7
32         this.classIL8 = class9CU; // Inheritance level: 8
33
34         this.classIL6String = IOUtils.fileToString(class7);
35         this.classIL7String = IOUtils.fileToString(class8);
36         this.classIL8String = IOUtils.fileToString(class9);
37     }
38
39     /**
40      * Let classes with inheritance level of 6 pass (6 < threshold).
41      * Uses
42      * {@link MeasureTest#skipIssue(com.github.javaparser.ast.Node, String)}
43      */
44     @Test
45     public void skipClassIL6() {
46         skipIssue(this.classIL6, this.classIL6String);
47     }
48
49     /**
50      * Report a problem for classes with inheritance level 7 (7 == threshold).
51      * Uses
52      * {@link MeasureTest#findIssue(com.github.javaparser.ast.Node, String)}
53      */
54     @Test
```

```
55      public void findClassIL7() {
56          findIssue(this.classIL7, this.classIL7String);
57      }
58
59      /**
60       * Report a problem for classes with inheritance level 8 (8 > threshold).
61       * Uses
62       * {@link MeasureTest#findIssue(com.github.javaparser.ast.Node, String)}
63       */
64      @Test
65      public void findClassIL8() {
66          findIssue(this.classIL8, this.classIL8String);
67      }
68
69      /**
70       * Used in superclass {@link MeasureTest} to determine if the correct
        issue
71       * was found.
72       */
73      @Override
74      public String getIssueType() {
75          return CISQMM06ClassInheritanceLevel.ISSUE_TYPE;
76      }
77  }
```

4.6.3 Iterations 12-15

The last phase of development was conducted during weeks 1 through 4, 2016. The main goal of this phase was to implement the remaining measures, providing a report of the QC_j quality score (see Section 2.4.7.4), and ending the development stage of the project. Additionally, a GUI providing the user a means to change settings was created (see Fig. 4.9), and CISQ maintainability problems were updated to be displayed as warnings (see Fig. 4.10 and Fig. 4.11). The QC_j quality score report was implemented with a simple custom Eclipse view (see Fig. 4.12). User stories 8 through 27 were completed during these iterations.

4.6.4 Iteration 16

After all CISQ measures were implemented, initial correctness- and performance tests were performed with SQUIDS running analysis on the five open-source projects discussed in Section 3.2.3.3. Iteration 16 was added primarily to fix bugs (programming errors) in the source code of SQUIDS which caused system crashes, incorrect results and slow analysis. These bugs were not detected before by the unit tests and running analysis of the example project. First, bugs which made the system crash when analyzing the projects were fixed. Afterwards, comparing the counts of

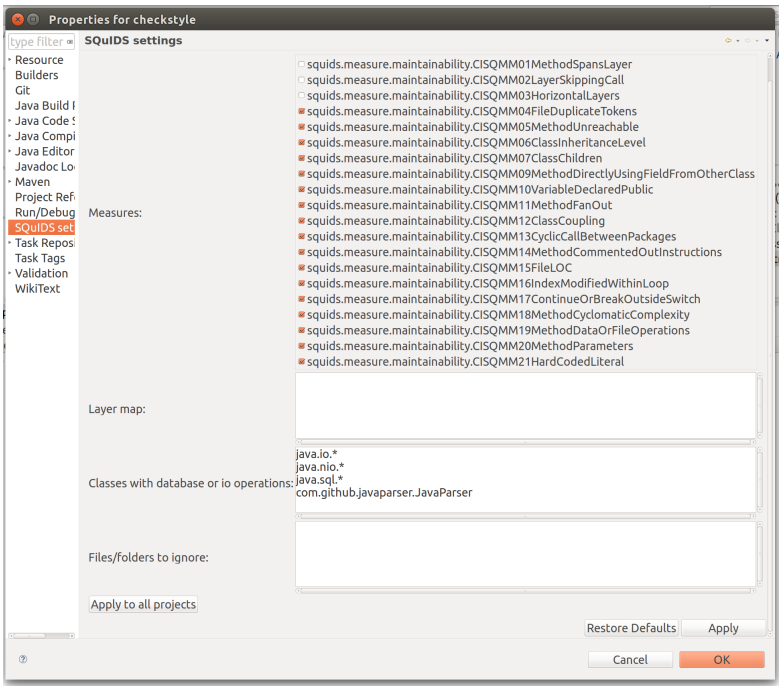

FIGURE 4.9: The SQUIDS settings-page displayed to the user in Eclipse.

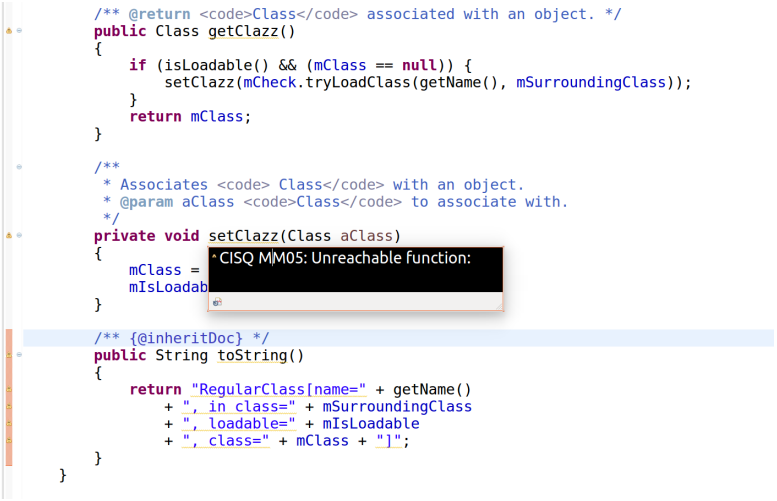

FIGURE 4.10: Eight CISQ maintainability problems displayed in Eclipse editor.

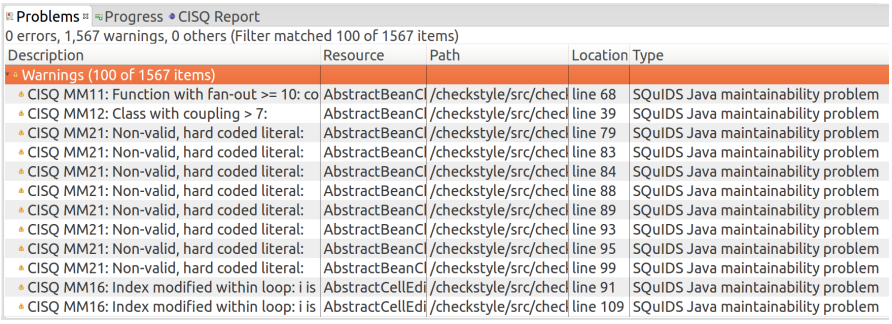

FIGURE 4.11: CISQ maintainability problems displayed in the Eclipse Problems view.

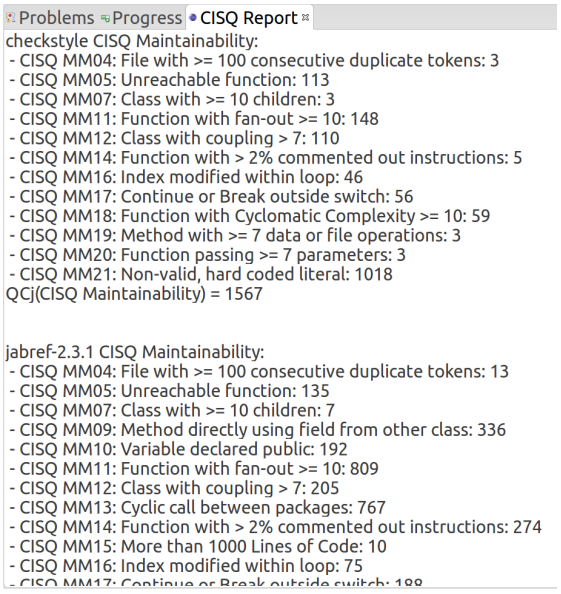

FIGURE 4.12: Report of violation counts displayed in a custom view called CISQ Report.

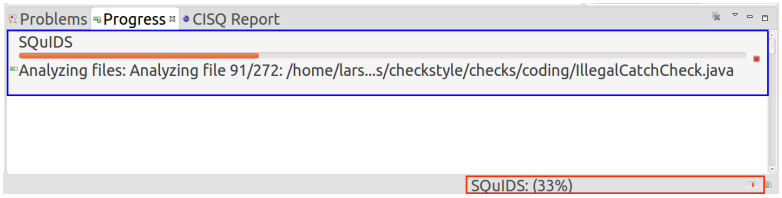

FIGURE 4.13: SQUIDS provides a percentage-done indicator in the bottom toolbar (red) and a progress-meter in the Eclipse Progress view.

problems reported for each CISQ measure with MUSE led a manual inspection of the CISQ measures which were considered as complex and had a very different result from MUSE. Both erroneously reported problems (false positives) and undetected problems (false negatives) were found, which led to improvements on the specific CISQ measures causing them. Finally, major performance improvements were made on CISQ measures which were exceptionally slow, such as M04 and M13.

Additionally, user story 28 was written and implemented, (see Appendix A) which provided a percentage-done indicator in the Eclipse Progress view and bottom toolbar when analyzing the project (see Fig. 4.13). This was added, as it became clear that four of the five open-source projects could not be analyzed within Nielsen's 10 seconds-limit (see Section 2.1.5) (Nielsen, 1993).

4.7 Chapter Summary

SQUIDS was created based on 19 of the 21 CISQ maintainability measures. Requirements were written based on user scenarios and the list of compliance requirements in the CISQ specification. The requirements and measures were then adapted to user stories, and the software was designed with context-, UML class- and BPMN diagrams in order to model environmental, structural and functional design.

Java and Eclipse were used together with the Eclipse PDE for development. The approach of using ASTs for source code analysis proved mostly uncomplicated. However, some weaknesses with the chosen tool (JavaParser) combined with some undefined terms in the CISQ specification caused development to slow down at times.

The resulting software is an Eclipse plug-in, with implementations for all the 20 applicable CISQ maintainability measures[3], which provides in-line problem markers in the source code editor, the Eclipse Problems view and a custom Eclipse view with a report of the quality score per project.

[3]M08 was not applicable to Java software (see Section 3.2.3.1).

Chapter 5

Results

This chapter presents the research results of this project, comprising implementation completeness of the CISQ Specification, performance numbers, unit test coverage and comparison of analysis results with the MUSE software. How the results meet the functional and nonfunctional requirements of SQuIDS (see Section 4.2) is also briefly presented. In this chapter, the term *measure* is used both for the measures of result analysis (e.g. performance measures, size measures etc.) and for the implemented CISQ maintainability measures. For clarity, when referring to the latter, the term *CISQ measure* is used.

5.1 Implementation

The resulting software artifact of this project is SQuIDS, an installable plug-in for the Eclipse IDE, meeting nonfunctional requirement #1 (see Section 4.2.2). It analyzes the source files of Java projects according to 20 of the 21 CISQ maintainability measures. This meets functional requirement #1 (see Section 4.2.1).

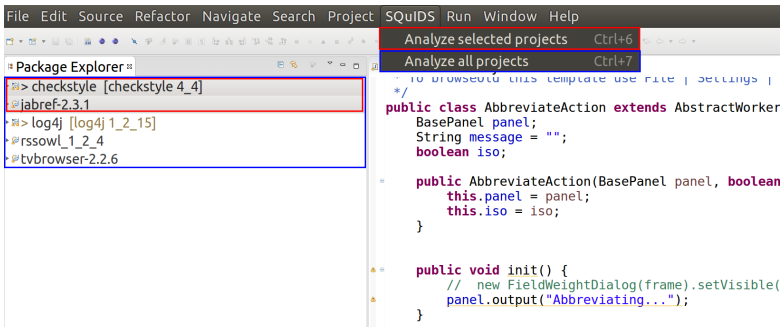

FIGURE 5.1: SQuIDS Commands: *Analyze selected projects* (red) and *Analyze all projects* (blue)

5.1.1 Functionality

It features two commands; One for analyzing all projects in the workspace, and one for analyzing just the selected projects (see Fig. 5.1). In addition, it provides a page in each project's properties-window (see Fig. 4.9), where settings can be altered for:

- which measures to include. The default setting is all implemented measures
- how the logical layers of the application's design are structured. Defaults to no layers. This meets functional requirement #2 (see Section 4.2.1)
- which classes or packages include data or file operations. Defaults to Java data and file packages `java.io.*`, `java.nio.*` and `java.sql.*`, and the `JavaParser` class from the JavaParser library which is known to contain file handling. This meets functional requirement #3 (see Section 4.2.1)
- which files and folders to ignore (e.g. test files)

However, most Java projects would not require additional configuration than installing the plug-in, due to the provided default settings, thus meeting nonfunctional requirement #2 (see Section 4.2.2).

Analysis of a Java project triggers a background process running SQUIDS, where CISQ maintainability problems are added to the list of warnings in the Problems view of Eclipse as they are found (see Fig. 4.11). The problems are marked with the line number where the problem occurs, and the source code is marked in the entire problem area (see Fig. 4.10). This meets functional requirement #4 (see Section 4.2.1) and nonfunctional requirement #3 (see Section 4.2.2).

5.1.2 Extensibility

In order to follow the fourth DSR guideline of contributing to both the community of scientific researchers and IT-professionals who facilitated the project (see Section 3.1.4), the API of SQUIDS is designed for extensibility in three ways:

1. Additional quality measures may be added by extending the `Measure` class (see Section 4.3.2 and Fig. 4.4), and existing CISQ measures may be turned on or off via the settings (see Section 5.1.1)
2. Additional information may be passed on to new measures, via the `settings` parameter in the `analyzeNode()` method in the `Measure` class.

3. The `IssueFinder` class and its dependencies, such as the `Measure` and `JavaParserHelper` classes are independent of IDE and operating system, and may therefore be used to implement similar plug-ins for other IDEs or tools

Therefore, nonfunctional requirement #4 is met (see Section 4.2.2). SQUIDS is however not designed to be extended for other programming than Java, languages, since as of now it highly depends on the JavaParser library.

5.1.3 CISQ Measures implemented

All CISQ maintainability measures were implemented, except for M08: *# of instances of multiple inheritance of concrete implementation classes (threshold* > 1*)*, which is obsolete for Java software (see Section 3.2.3.1).

5.1.4 CISQ Compliance Requirements met

The CISQ specification's list of required attributes and inputs (see Section 2.4.7.3) were followed to the extent of a partial implementation of the specification (only the maintainability measures).

5.1.4.1 Automated

SQUIDS is fully *automated* for all CISQ measures except M02, M03, which require a description of the architectural layers in the project, and M19, which requires a list of classes or packages containing data or file operations. As stated by the specification, this is allowed.

5.1.4.2 Objective

Two independent analyses by SQUIDS of a Java project produces the same result for each CISQ measure each time. SQUIDS does not rely on fuzzy logic, which means that a given input (source code) will always result in exactly one output.

5.1.4.3 Transparent

Each required configuration input is listed in the project properties-window (see Fig. 4.9), which, along with the source code visible in the Eclipse IDE, confines the list of required input for analysis. The output (problems found) is listed in the Problems-view (see Fig. 4.11) and visible in the Editor-view in Eclipse (see Fig. 4.10).

5.1.4.4 Verifiable

All implemented CISQ measures are commented with Javadoc, where the assumptions and considerations used in analysis are explained.

5.1.4.5 Required Inputs

As stated by the specification, both the entire source code of, and the information about architectural layers in, the application being analyzed is required. The last four required inputs are however not required in SQUIDS, as they would be redundant in a partial implementation (only the maintainability characteristic).

5.2 Analysis Results

The final SQUIDS analysis results of the five open-source Java projects is displayed in Table 5.1. Each row displays the counts of CISQ rule violations found by an implemented CISQ measure for each of the open-source Java projects (see Section 3.2.3.3). Based on the **Total**-row, which is equal to the QC_j score of the maintainability characteristic for each project (see Section 2.4.7.4), the following list ranks the projects from most maintainable to least:

1. Checkstyle 4.4
2. Log4j 1.2.15
3. TV-Browser 2.2.6
4. JabRef 2.3.1
5. RSSOwl 1.2.4

CISQ measure M06 found the smallest count of maintainability problems: a single instance of a class in the TV-Browser project which had an inheritance level of seven or higher (see Table 2.4). M21 reported a large amount of problems, totaling on 60,496 reported instances of hard-coded literals.

TABLE 5.1: SQUIDS results from analysis of the five open-source Java projects.

	Rule violations				
CISQ Measure	**Checkstyle 4.4**	**JabRef 2.3.1**	**Log4j 1.2.15**	**RSSOwl 1.2.4**	**TV-Browser 2.2.6**
M04	3	10	2	4	13
M05	113	134	68	82	251
M06	0	0	0	0	1
M07	3	7	1	3	3
M09	0	336	17	540	45
M10	0	192	4	175	38
M11	148	808	162	524	1,065
M12	110	204	24	53	77
M13	0	767	104	1,329	805
M14	5	274	29	5	56
M15	0	10	1	3	2
M16	46	75	7	3	94
M17	56	188	21	53	110
M18	59	215	23	114	178
M20	3	11	4	4	28
M21	1,018	12,046	1,578	34,432	11,422
Total	1,564	15,277	2,045	37,324	14,188

5.3 Performance

The performance of SQUIDS was measured by the time used for analysis by each of 16 CISQ measures. The remaining four implemented CISQ measures, M01, M02, M03 and M19 were not used, as they require manual configuration of the architectural layers and data or file operations used in the projects. This configuration would require intricate knowledge about the projects, which was not available.

5.3.1 Setup Details

Time was measured by getting the system time directly before and after each call to the method `analyzeNode()` for each CISQ measure, and summing it up after analysis was complete (see Snippet 5.1). This was done instead of simply measuring the time taken for each project, to measure the performance of each CISQ measure:

Snippet 5.1: Getting the time used by each CISQ measure

```
1  // Get the time before the call
2  long startTime = System.currentTimeMillis();
3
4  // Perform the call
5  List<Issue> measureIssues = measure.analyzeNode(rootNode, fileString,
       this.compilationUnits);
6
7  // Get the time after the call
8  long stopTime = System.currentTimeMillis();
9  // The difference is the time taken by this CISQ measure for this node
10 long elapsedTime = stopTime - startTime;
11 // Add the time taken to the total time taken by the CISQ measure
12 addTimeToMeasure(measure, elapsedTime);
```

This measurement was done each time on the same computer, for each of the five open-source Java projects (see Section 3.2.3.5), with some other active programs and services to simulate a normal running environment for SQuIDS (see Table 3.4).

5.3.2 Measures

The measures used for calculating the performance on each of the five open-source Java projects were:

- Size: Number of Java files, physical LOC (PLOC) and average PLOC
- Time per CISQ measure: Seconds used and seconds used per 10 files
- Problems: The number of problems found per CISQ measure

5.3.3 Performance Test Results

The results from the performance test (see Table 5.4) revealed that the implemented measures were approximate to the time limit of 1 second per 10 files in nonfunctional requirement #5 (see Section 4.2.2). However, as can be seen in Table 5.2 and Table 5.3, the time SQuIDS uses does not increase linearly with size, which the requirement specified. Additionally, the average total time of analysis was 2.310 seconds, which does not meet the requirement. Log4j, with 168 Java files and a total of 20,784 PLOC, took 4.054 seconds to analyze, while RSSOwl, with 201 Java files and about 3 times the total PLOC as Log4j, took 79.634 seconds - more than 19 times as long. It is evident that the number of problems found has a high impact on the performance. RSSOwl proved to have about 18 times as many problems as Log4j.

TABLE 5.2: SQUIDS performance for each of the five open-source projects.

Project	Java files	Total PLOC	Problems	Total time
Log4j 1.2.15	168	20,784	2,068	4.054
Checkstyle 4.4	272	30,827	1,601	9.176
TV-Browser 2.2.6	653	86,903	14,536	33.200
JabRef 2.3.1	421	60,564	15,851	53.226
RSSOwl 1.2.4	201	65,540	37,554	79.634

TABLE 5.3: SQUIDS performance for each of the five open-source projects per 10 files.

Project	Java files (total)	PLOC per 10 files	Problems per 10 files	Time per 10 files (s)
Log4j 1.2.15	168	1,237.14	123.10	0.299
Checkstyle 4.4	272	1,133.35	58.86	0.250
TV-Browser 2.2.6	653	1,330.83	222.60	0.658
JabRef 2.3.1	421	1,438.57	376.51	1.413
RSSOwl 1.2.4	201	3,260.70	1,868.36	4.582

Exceptionally fast CISQ measures are M06, M10, M17 and M20, at around 2 milliseconds per 10 files (see Table 5.4), where the average is 65 milliseconds. Notably slow CISQ measures are M04, M12 and M21, taking between 259 and 424 milliseconds per 10 files. The differences may be partially attributed to the number of problems found by the CISQ measures (see Table 5.1), but the slow M04 found a small amount of problems. A larger factor for the performance of M04 (and others) is that it simply requires complex analysis in order to find problems.

TABLE 5.4: Average SQUIDS performance from the five open-source projects. (see Appendix D for detailed results per project).

CISQ measure	Time used (s)	Time per 10 files (s)
M04	10.675	0.311
M05	0.465	0.014
M06	0.026	0.001
M07	3.797	0.111
M09	0.662	0.019
M10	0.079	0.002
M11	0.949	0.028
M12	3.828	0.112
M13	0.752	0.022
M14	0.217	0.006
M15	0.227	0.007
M16	0.716	0.021
M17	0.086	0.003
M18	2.022	0.059
M20	0.023	0.001
M21	16.749	0.488
Measure time[a]	41.275	1.203
Parsing time	0.755	0.022
Marking time	35.703	1.041
Other operations	1.513	0.044
Total time	79.245	2.310

[a]Total time used by the CISQ measures

5.4 Unit Test Coverage

The unit tests written for SQUIDS follow the principle of Black Box testing (see Section 2.1.8). Each CISQ measure has a dedicated test class which subjects it to different kinds of inputs, and asserts that the output is as expected. For CISQ measures with threshold values, e.g. M11: *# of functions that have a fan-out* $\geq$ 10, boundary values are tested with three test cases, meaning that the CISQ measure is subjected to fan-out counts of 9, 10 and 11 (see Section 2.1.8.2), and a problem reported by M11 should be found when subjected to either 10 or 11, but not 9.

TABLE 5.5: Line- and Mutation Coverage for unit tests created for implemented CISQ maintainability measures

CISQ Measure	Line Coverage	Mutation Coverage
M01	100%	75%
M02	100%	96%
M03	100%	80%
M04	100%	68%
M05	92%	87%
M06	100%	83%
M07	100%	85%
M09	90%	79%
M10	100%	86%
M11	99%	87%
M12	96%	81%
M13	95%	63%
M14	93%	79%
M15	100%	100%
M16	97%	82%
M17	100%	86%
M18	94%	85%
M19	100%	92%
M20	100%	100%
M21	92%	77%
SuperClass1[a]	100%	100%
SuperClass2[b]	99%	82%
SuperClass3[c]	100%	0%
Total	97%	82%

[a]Class `CISQMMLayerDependentMeasure` is the superclass of CISQ measures M01, M02 and M03.
[b]Class `CISQMMTypeDependentMeasure` is the superclass of CISQ measures M05, M09, M11, M12, M13, M16, M19 and M21.
[c]Class `CISQMaintainabilityMeasure` is the superclass of CISQ measures M04, M06, M07, M10, M14, M15, M17, M18, M20, `CISQMMLayerDependentMeasure` and `CISQMMTypeDependentMeasure`

The unit test requirements in the test design were met (see Section 3.2.3.2). Which types of input to test was never formalized, and additional test cases for the CISQ measures were made on a per CISQ measure basis, making the test design semi-formal. The 20 implemented CISQ measures have a total of 164 unit test cases, of which none fail. The line coverage analysis and mutation tests performed with PITEST resulted in a total line coverage of 97%, where no CISQ measure had a line coverage below 90% (see Table 5.5), which follows the 4^{th} unit test requirement. The total mutation coverage was 82%.

5.5 Chapter Summary

SQuIDS implements the CISQ maintainability characteristic to specification, although one of the maintainability measures (M08) was ignored due to redundancy (see Section 5.1.3). The compliance requirements of the CISQ specification are met for a partial implementation of the specification (only the maintainability characteristic). The required inputs for analysis are configurable in a visual editor of project properties, and the required outputs are visible as a list of problems found, and as markings directly in the source code editor in the Eclipse IDE.

The average performance of the CISQ measures in SQuIDS is close to the required speed, superseding the limit of less than 1 second with 200 milliseconds for analysis of 10 files. The average total time of analysis was unfortunately 2.310 seconds, thus not meeting the requirement. Instead of a linear complexity, where the double amount of files should only take the double amount of time to analyze, the complexity is exponential.

The 164 unit test cases developed for SQuIDS are semi-formal Black Box tests written for each implemented CISQ measure, where border values for thresholds are always analyzed.

All functional and nonfunctional requirements were met, with the exception of nonfunctional requirement #5, where the performance of SQuIDS failed to meet the target speed and complexity.

Chapter 6

Discussion

This chapter discusses the methods and theory used in the project, including the CISQ specification, and challenges the implementation of the software artifact SQuIDS. Following, the research questions are answered, and finally, the validity of this study is examined.

6.1 Discussion of Methods

Using the DSR framework as a research method proved suitable for this project, which was expected when the project's goal was to create and describe the development of an IT artifact which solves a specific task (Hevner et al., 2004).

The seven guidelines of DSR were followed throughout research and development. Specifically guidelines 2-4 were helpful in both research and development. Guideline 2: *Problem relevance*; and 3: *Research contributions* made an impact on development. The focus on creating an implementable solution which contributes to the software quality community increased the work on the expandability and usability of the API of SQuIDS, even though those traits were not essential to the research questions. Guideline 3: *Design evaluation* provided a categorization of available evaluation techniques, where three of them were chosen for evaluating SQuIDS.

6.1.1 Size Measures

The size measures used in the dynamic analysis of SQuIDS's performance do not by themselves provide an exact prediction of performance, as the number of problems found was a highly influential factor. For example, the TV-Browser project had a larger count of total PLOC than the RSSOwl project (see Table 5.2), and over three times the number of files, but the RSSOwl project took more than twice as long as

TV-Browser to analyze. A reasonable explanation is that the number of problems found in RSSOwl has a high impact on the time it takes to complete analysis: RSSOwl has over twice the number of problems as TV-Browser. This relation may not be the same for all cases, such as with the projects Checkstyle and Log4j, where Checkstyle has roughly 3/4 the number of problems as Log4j, but took more than twice the time to analyze. Nonetheless, the impact of problems present in a project is notable.Table 5.3 shows that the number of Java files, total count of LOC, and problems found in a project, are all variables which influence the performance of SQuIDS.

6.1.2 Performance Test Amount

Performance test results are gathered from the average of three single analyses per project, to ensure that the results were less dependent of system (computer) status at test time. For example, if the computer was receiving updates at the exact time when a certain project was being analyzed, that project would take longer to analyze, as the completion time of any task in a computer is influenced by the amount and complexity of other simultaneous tasks. The amount of analyses performed per project may be debatable, though the differences between the analyses were small enough to assume that additional tests would not add more information.

6.1.3 Performance Test Points

Performance measuring was divided between the time taken by the CISQ measures, the time taken by the JavaParser library to parse the source files, the time taken by Eclipse to mark the problems in the files, and the rest of the SQuIDS logic. While the total time it takes to analyze a project is the focus of the performance measuring of SQuIDS, this separation was done to find which parts of the application that are most time consuming. This way, certain parts of the application may be targeted for performance improvement with a specific solution, thus supporting future development. For example, exceptionally slow CISQ measures could be targeted for improvement. Additionally, the time it takes JavaParser to parse all the source files may be decreased by not parsing each file in the project anew for each analysis, and instead keep track of all changes made since last analysis, and simply re-parse the changed files.

6.1.4 Unit Test Formality

The test technique applied in unit testing of SQUIDS was kept semi-formal (see Section 3.2.3.2 and Section 2.1.8). Keeping a more formalized test technique could have resulted in a larger test coverage. For example, if the test technique specified that each (subclass of) `Measure` should be tested for each method, and given each possible input, line coverage and mutations killed in mutation testing could be improved (see Section 5.4). Unfortunately, this would cost a substantial amount of time, which this project could not afford. However, a total line coverage of 97% and a mutation coverage of 82% can be considered as acceptable for this project.

6.1.5 Informed Argument

Making an informed argument on the basis of comparing results with Plösch, Schürz, and Körner (2015) has some weaknesses. The CISQ measures implemented in the MUSE software and their results are not verified by any third party. Therefore, a comparison does not provide a view of how *correct* SQUIDS is, but rather how similar it is to another implementation.

6.2 Discussion of the CISQ Specification

There are a few concerns with the CISQ specification which may impact the validity of an implementation. These issues will be exhibited in this section.

6.2.1 Ambiguous Terms

The ambiguous terms in the CISQ specification (see Section 4.5.1) introduces a problem with the correctness of the implementation. If a term in the specification is misunderstood, the implementation may produce results which were not intended by the specification. For example, if the term *cyclomatic complexity* in the CISQ specification does not refer to McCabe's Cyclomatic Complexity (1976), SQUIDS may calculate the cyclomatic complexity of a method differently than what was intended by the specification (see Section 2.4.6). On the other hand, relying on the assumptions on some of the terms made by Plösch, Schürz, and Körner (2015), and specifying the calculation techniques implemented in SQUIDS both in the source code and this thesis, the external validity of the project is maintained on this point.

6.2.2 Source Code Size Metrics

The only size metric in the CISQ specification is the LOC of each file. While the specification states that the quality measure elements (metrics) in the specification were selected due to high impact (CISQ, 2012, p. 12), the total system size is not covered by the specification. If the LOC of each file in a project is under 1,000, the CISQ specification would still not consider it to be a problem, even if the total LOC in the project is in the number of millions. On the other hand, a large total LOC of a project is considered to be a problem (Nguyen et al., 2007; Sjøberg, Anda, and Mockus, 2012). Implementing another measure which can detect this problem would be of little difficulty, as the already implemented CISQ measure M15: # *of files* $>$ 1000 *LOC* could be used to sum the total LOC of a software project.

6.2.3 Improving Maintainability

In contrast to concluding that the CISQ specification could be used for quality evaluations, Plösch, Schürz, and Körner (2015) argue that its coverage of aspects of maintainability deems it only "partially suitable for improvement programs" compared to their own EMISQ model (see Section 2.4.10). In other words, it is less suitable for recommending improvements to a software, because it may fail to identify numerous problems related to maintainability. Scenario 1 (see Section 4.1.1) is therefore threatened by this claim, while scenario 2 (see Section 4.1.2) is strengthened. On the other hand, scenario 1 requires SQuIDS to run at an acceptable speed, which is difficult if there are too many CISQ measures analyzing the code. On that matter, it is possible to say that the CISQ specification is still suitable for controlling maintainability during development, but is not recommended for improving existing code.

6.3 SQuIDS Implementation

There are primarily two subjects related to the implementation of SQuIDS which need further elaboration: One of the CISQ maintainability measures was ignored, and while the recognized programming pattern Visitor could have simplified the API of SQuIDS, it was not used.

6.3.1 Ignored Measure

While CISQ measure M08 was ignored in the implementation of the specification (see Section 3.2.3.1), an implementation could still have been made. The JavaParser library representation of a class or interface is the class `ClassOrInterfaceDeclaration`, featuring the method `getExtends()`, which provides a list of superclasses. Providing a list is meant for interfaces, which may extend multiple interfaces[1], though a CISQ measure could be developed to look for objects of the class `ClassOrInterfaceDeclaration` which are not interfaces (using the method `isInterface()`) and have an extension list containing more than one element. However, an actual implementation was never made because of testability: It is not possible for JavaParser to parse a class which has multiple superclasses[2]. In other words, an implementation could be made, but tests could not have been made to ensure correctness.

6.3.2 Not Using the Visitor Pattern

The `Measure` class does not use the Visitor pattern which is often used for traversing ASTs (see Section 2.4.1). It could very well have been used, as the JavaParser library allows for AST nodes to be accessed by classes extending the `GenericVisitorAdatper<R,A>` class by calling the `accept()` method on the node, and providing the CISQ measure as a parameter value (see Snippet 6.1).

Snippet 6.1: Hypothetical use of Visitor pattern

```
1 for(Measure cisqMeasure : measures) {
2     // The node accepts a class which extends
3     // GenericVisitorAdatper<R,A> and returns
4     // a list of problems. Additional info may
5     // be added as the second parameter.
6     List<Issue> issues = node.accept(cisqMeasure, info);
7 }
```

The implementation required the string representation of the file being analyzed in order to correctly calculate the position of the node (see Section 4.5.2). Additionally, a complete list of the other files in the project was required in order to implement measures which regard relations between files (e.g. M06, M07 or M09). The `accept()` method which in turn calls the `visit()` method on the provided visitor accepts a

[1] Interfaces use the `extends` keyword to extend multiple interfaces. Therefore, the `getExtends()` method must retrieve a list, as the `ClassOrInterfaceDeclaration` class does not discriminate between the two. However, a `getImplements()` method is also available, which is redundant for interfaces.

[2] Attempting to parse a file with a class with multiple superclasses with JavaParser throws the following exception: `com.github.javaparser.ParseException: A class cannot extend more than one other class`

single additional parameter, which could be used to add the string representation and list of other files, by packaging them into another object (see Snippet 6.2).

Snippet 6.2: Hypothetical CISQ measure using Visitor pattern

```
1  class CISQMeasure extends
2              GenericVisitorAdapter<List<Issue>, Map<String, Object>> {
3
4          @Override
5          public List<Issue> visit(Node node, Map<String, Object> info) {
6              List<Issue> issues = new LinkedList<>();
7
8              // Get the string representation of
9              // the file being analyzed.
10             String fileString = (String) info.get("file");
11
12             // Get the list of other files (as
13             // CompilationUnits).
14             List<CompilationUnit> otherFiles = (List<CompilationUnit>)
15                     info.get("comp_units");
16             // ...
17             // Analyze the node and return issues
18             // if problems are found.
19             // ...
20             return issues;
21         }
22     }
23 }
```

This was, unfortunately, only discovered late in development and was dropped, as refactoring to the Visitor pattern would require too much effort close to the deadline. Snippet 6.1 and Snippet 6.2 do however display the opportunity of improving SQUIDS by refactoring to the Visitor pattern, which would "simplify what is complicated, and [...] make [the] code better at communicating its intention" (Kerievsky, 2004, p. 6).

6.4 Comparison of Results with MUSE

The raw results from the MUSE software (Plösch, Schürz, and Körner, 2015), provided by Schürz (2016) revealed that their implementation of the CISQ measures gives a result with many differences from SQUIDS (see Table 6.1). While there were CISQ measures with equal amounts of problems found between the two implementations, the analysis of the TV-Browser project shared no correspondence with the analysis by MUSE (see Appendix E.5), which may be caused by different versions of the project in the two result sets. Therefore, the TV-Browser project is omitted in the totals table (see Table 6.1). Tables for each project, and totals with the TV-Browser project included are available in Appendix E.

TABLE 6.1: Comparison of total numbers of problems found per CISQ measure between SQuIDS and MUSE (Schürz, 2016), without the TV-Browser project.

CISQ measure	SQuIDS count	MUSE count	Difference
M04	19	0	19
M05	397	47	350
M06	0	17	17
M07	14	14	0
M09	893	1,712	819
M10	371	371	0
M11	1,642	1,150	492
M12	391	766	375
M13	2,200	183	2,017
M14	313	600	287
M15	14	31	17
M16	131	16	115
M17	318	241	77
M18	411	237	174
M20	22	22	0
M21	49,074	49,562	488
Total	56,210	54,969	1,241

CISQ measures M07, M10 and M20 produced the same number of problems in MUSE and SQuIDS. M21 found a similar number of problems in each implementation, where SQuIDS found 49,074 and MUSE 49,562. M09, M11, M12, M14, M15, M17 are more different, while the last five, M04, M05, M06, M13 and M16 are proportionally the most different CISQ measures. M04 and M06 are points of interest, where MUSE found no problems for M04, and SQuIDS found 22 problems, and vice versa for M06.

6.4.1 Detailed Comparison

By manually inspecting and comparing the maintainability problems which SQuIDS found and MUSE did not, and vice versa, major differences between some implementations of the CISQ measures were found. The comparison was made possible by creating XML reports[3] with the same format as the raw results from MUSE, and finding the exact locations of problems which were only reported by one analyzer, using a program developed for this specific purpose[4]. The program searches the XML reports from SQuIDS and MUSE for problems which are only reported by one of the implementations. This is done by creating IDs for the problems, consisting of the CISQ measure, file number and line number (e.g.

[3]XML reports were created with the `XMLWriter` class

[4]The program used for finding problems reported by only SQuIDS or MUSE is available at https://github.com/larsac07/CISQAnalyzerComparator

`M06:JABREF_ROOT/src/java/net/sf/jabref/FileHistory.java:9`), and excluding problems with IDs which exist in the reports of both implementations. The remaining problems are only reported by one of them.

Where a detailed review of each problem which was only reported by one of the implementations would require too much effort for the scope of this thesis, the comparison was limited to a small sample of CISQ measures. Assumptions about the other CISQ measures are available in Appendix B. M04 was selected to look at why SQuIDS found 19 problems for M04, and MUSE none. M06 was selected for the same reason, where SQuIDS did not find any problems for M06, whereas MUSE found 17. Additionally, M17 was selected because the difference between the number of problems found by SQuIDS and MUSE was relatively low, and considering it is easy to determine whether a problem is valid (true positive) or mistakenly reported (false positive), by manual inspection of the source code. Notes from the manual inspection are available in Appendix F.

6.4.1.1 Problems for M04

SQuIDS was the only implementation which found problems for M04. Additionally, the way SQuIDS reports problems for M04 makes it hard to manually inspect results: In M04, SQuIDS converts each file to a sequence of tokens (see Section 2.4.1), and looks for duplicates. Duplicates are reported, but exact locations are not calculated. M04 should therefore be improved before performing a manual comparison with another CISQ implementation. Why MUSE did not find any problems for M04 is unknown, but may be attributed to a different interpretation of the specification or another definition of tokens.

6.4.1.2 Problems for M06

All the problems reported for M06: *# of classes with inheritance levels* ≥ 7 were found in classes which either directly or indirectly extend classes from the Java Swing package (`javax.swing`) or the JGoodies library (`com.jgoodies.uif_lite.panel.SimpleInternalFrame`). The problems were only reported by MUSE (SQuIDS found no problems), and each problem counted an inheritance level of exactly 7. If the inheritance level is counted locally, i.e. only inheritance from the classes in the project is counted, none of the classes which were reported by MUSE have an inheritance level of 7 or higher. Contrarily, if the inheritance from classes outside the project is counted, all the reported classes have an inheritance level of exactly 6. This is due to some of the classes in the Swing package

having a high inheritance level to start with, such as `JMenu`, which has an inheritance level of 5. If the `Object` class is counted, the reported classes have an inheritance level of 7. The CISQ specification has been interpreted differently by the two implementations.

6.4.1.3 Problems for M17

Between the two implementations, there were 225 identical problems reported for M17: *# of GO TOs, CONTINUE, and BREAK outside the switch*. There were no problems for M17 reported by MUSE which SQuIDS did not also report. On the contrary, there were 93 problems which only SQuIDS reported. The reason this number is higher than the difference of 77 (see Table 6.1) is that the manual inspection revealed that MUSE sometimes report duplicate problems. For example, the break statement on line 182 in the class `ExecutableStatementCountCheck`[5] in the Checkstyle project is reported twice. Continuing on the 93 problems only detected by SQuIDS, certain conditions around the findings were noted. The problems which MUSE did not report were one or more of the following:

- `break` or `continue` with a label
- `break` or `continue` which is located within an `else if`-statement
- `break` or `continue` which is located within an `if`-statement which in turn is located within a `switch`-statement
- `break` or `continue` which is located within an `if`-statement in a final or anonymous inner class

On another note, SQuIDS reports 4 instances of a `break` with a label, within one or more `if`-statements, within a `switch`-statement with the same label, thereby making it explicit that the `break` belongs to the `switch`, and not a surrounding loop (see example in Snippet 6.3). In the example in Snippet 6.4 below, no label is provided. This makes the segment less readable, as the relation of the break is not explicit, and must be decoded and understood by the reader. It is debatable and arguably a question of definition whether this is the case for the labeled example as well. The CISQ specification provides no further explanation, but it can be argued that even when a label is present, the code is over-complicated due to surrounding a `break` inside an `if`-statement.

[5] Located at `CHECKSTYLE_ROOT/src/com/puppycrawl/tools/checkstyle/checks/sizes/ExecutableStatementCountCheck.java`

Snippet 6.3: Break with a label inside an if inside a switch with the same label.

```
for (int i = 0; i < array.length; i++) {
    label_1: switch (variable) {
        case 1:
            if (condition) {
                break label_1;
            }
        // ...
    }
}
```

Snippet 6.4: Break inside an if inside a switch.

```
for (int i = 0; i < array.length; i++) {
    switch (variable) {
        case 1:
            if (condition) {
                break;
            }
        // ...
    }
}
```

6.5 Answering the Research Questions

Having thoroughly discussed the methods, theory and implementation of the CISQ specification, the research questions may be answered using the research design described in Section 3.2, and the results presented in Chapter 5.

6.5.1 Research Question 1

How can a static code analyzer be developed as an Eclipse plug-in which, based on a standard, detects maintainability problems in a software project?

The software artifact SQUIDS implements the maintainability characteristic of the *CISQ Specifications for Automated Quality Characteristic Measures*, which conforms to the ISO/IEC 25010 standard. As concluded by Plösch, Schürz, and Körner (2015), the maintainability measures in the specification are possible to automate in a software. Unlike their MUSE software, SQUIDS is developed as an open-source, installable plug-in for the Eclipse IDE.

Following are the key technologies which together comprise the implementation of SQuIDS:

- The JavaParser library (see Section 4.4.2.2)
- AST traversal (see Section 2.4.1)
- The CISQ specification (see Section 2.4.7)
- The Eclipse PDE (see Section 4.4.3)

The cornerstone of the functionality in SQuIDS is using the JavaParser library to create ASTs of the source code. While traversing the ASTs, each tree node is analyzed by each CISQ measure where maintainability problems may be found. Combining the API of JavaParser with a set of helper methods (see `JavaParserHelper` in Section 4.5.2) and the Eclipse PDE, provides the functionality necessary to implement a static code analyzer for maintainability measures, based on the CISQ specification.

In Chapter 4, the design and development of SQuIDS describes how it was implemented.

6.5.2 Research Question 2

How can such an implementation visualize maintainability problems in a software project to the user?

The maintainability problems found by SQuIDS are reported in three locations in the Eclipse IDE:

- The source code editor (see Fig. 4.10)
- The Problems-view (see Fig. 4.11)
- A custom Eclipse-view called CISQ Report (see Fig. 4.12)

Problems are marked using Eclipse's `IMarker` API, which is accessed by creating an `IMarker` from an `IFile`, and setting values for predefined attributes (see Snippet 6.5). Eclipse then takes this information, and adds markers to the source code editor (see Fig. 4.10) and the Problems-view (see Fig. 4.11).

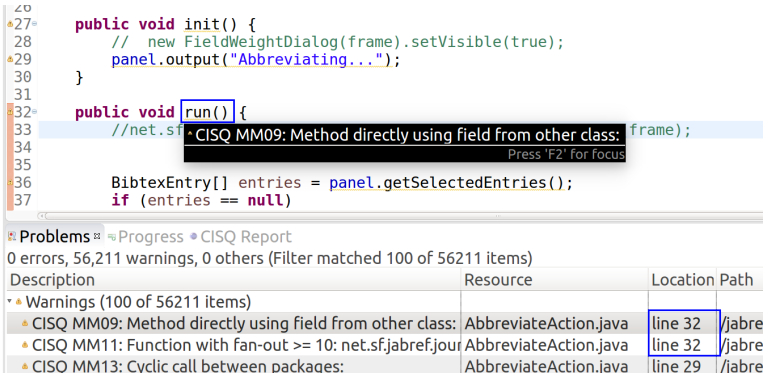

FIGURE 6.1: Two different problem markers are overlapping in the Eclipse source code editor.

Snippet 6.5: Using the `IMarker` API

```
1 // ...
2 IMarker m = iFile.createMarker("SQuIDS.javaqualityissue");
3 m.setAttribute(IMarker.LINE_NUMBER, errorLineNumber);
4 m.setAttribute(IMarker.MESSAGE, message);
5 m.setAttribute(IMarker.PRIORITY, IMarker.PRIORITY_NORMAL);
6 m.setAttribute(IMarker.SEVERITY, IMarker.SEVERITY_WARNING);
7 m.setAttribute(IMarker.CHAR_START, startIndex);
8 m.setAttribute(IMarker.CHAR_END, endIndex);
```

6.5.2.1 Source Code Editor

In the source code editor, overlapping problems have proven to be an issue (see Fig. 6.1), where the problem reported last covers the other, making it invisible. One solution could be to gather overlapping problems into single problems which contain lists. A different approach is to create another custom view of the problems, which could contain a list of problems where selecting a problem highlights the problem area in the source code editor.

6.5.2.2 Problems View

While the Eclipse Problems-view is the standard location for viewing errors and warnings (see Fig. 4.11), it may not be optimal for displaying maintainability problems. It has been noticed during testing that a large amount of problems makes it difficult for the view to display them. A large amount of problems makes the view slow, and may cause the entire IDE to freeze when scrolling through the list. As an alternative, a customized view for maintainability problems could be created, where optimization on computer memory and processor usage could be prioritized.

6.5.2.3 Usability Tests Recommended

Common for these issues, is that identifying optimal manners of displaying this kind of information to the intended users is needed. Further research and development on the project should therefore involve usability testing to improve on this part of the GUI, as well as the rest. One way of presenting maintainability problems in an IDE has been presented in this thesis, while other alternatives could be better. Yet, it is a proof-of-concept on displaying maintainability problems found by a plug-in in an IDE, using standard, provided tools.

6.5.3 Research Question 3

Can such an implementation give a performance which makes it usable during development?

The speed of analysis in a software which follows the structure of SQuIDS mainly depends on variables which can be identified before analysis. These variables are the amount and complexity of measures, and the size of the project(s) being analyzed. In addition, a variable which is harder to identify on beforehand, is the amount of problems to be found in the software to be analyzed, which also proved to make an impact on the analysis performance.

Dividing the stopwatch-approach of the performance testing into specific parts of SQuIDS (see Section 3.2.3.5) provided an indication of which variables affected which parts of the system, and how. First, a repetition of the different times which were recorded:

- Parsing: The time it takes the JavaParser library to parse all the source files in a project
- Analysis: The time it takes each CISQ measure to analyze all the source files in a project
- Marking: The time it takes to mark all the problems found in a project in the Problems-view and editor of Eclipse
- Rest: The rest of the time it takes to analyze a project (Total time minus the above)
- Total: The total time it takes to analyze a project

A repetition of the variables affecting the performance of SQuIDS:

TABLE 6.2: The time SQUIDS uses to analyze each of the five open-source projects.

Project	# of CISQ measures	Total PLOC of source files	# of problems found	Total analysis time (s)
Log4j	16	20,784	2,068	9.009
Checkstyle	16	30,827	1,601	11.108
JabRef	16	60,564	15,851	84.089
TV-Browser	16	86,903	14,536	103.181
RSSOwl	16	65,540	37,554	188.839

- Amount of CISQ measures
- Total LOC of source files
- Amount of problems found
  - The types of problems found

The performance test results (see Section 5.3) showed that the amount of CISQ measures used for analysis and the total LOC of the source files affect the time used for analyzing files and marking problems. Especially slow CISQ measures make a great impact on analysis. The time used on parsing the source files is directly connected to the total LOC of the source files being analyzed: The more code to parse, the more time it takes. The amount of problems to be found in the software being analyzed is directly responsible for the time used on marking the problems. Due to the structure of SQUIDS, that amount also influences the time used on analysis, as each implemented CISQ measure uses time on reporting the problems found. In addition, the types of problems to be found affects both analysis and marking. If the software being analyzed contains a large amount of a problem which is found by a slow-working CISQ measure, this may have a significant impact.

To answer this research question, results from analyzing the five open-source test projects and the performance tests are be used together in Table 6.2 to indicate which velocities can be expected from a software such as SQUIDS. With only one of the projects (Checkstyle) analyzed below the 10-second limit for keeping the user's attention to the dialog (see Section 2.1.5), it is reasonable to conclude that when analyzing larger projects, the attention of the user will be lost. However, a progress indicator with percentage-done is displayed to the user (see Fig. 4.13), so that other tasks may be attended to in the meantime. With an average analysis time of ≈ 79 seconds for the five test projects, the developer may still use SQUIDS multiple times during a workday without making a large impact on the amount of time dedicated to programming. This is without calculating the work effort saved later by removing maintainability problems found with the software early.

6.6 Validity

The resulting software artifact and its test results mainly corresponds to expectations and the intention of the research design of this project. Nevertheless, there may be weaknesses in this project, either internally, or for external purposes.

6.6.1 Internal Validity

The internal validity concerns relate to providing a proper answer for Q2, and the correctness of the implemented CISQ Measures.

6.6.1.1 Properly Answering Q2

Although Q2 only requires an example of how maintainability problems can be displayed (see Section 1.3), this study did not explore multiple solutions, but rather developed one solution, and committed to it. According to Hevner et al. (2004), this is not problematic, as DSR is intended to find a satisfactory solution, "without explicitly specifying all possible solutions". Still, as discussed in Section 6.5.2, the evaluation the solution of Q2 could be improved with usability tests.

6.6.1.2 Correctness of Implemented CISQ Measures

It cannot be guaranteed that the 20 implemented CISQ maintainability measures in SQUIDS are 100% correct implementations of the CISQ specification, due to three reasons:

- The specification contained ambiguous terms
- SQUIDS produces different results from the MUSE software developed by Plösch, Schürz, and Körner (2015)
- The results from Plösch, Schürz, and Körner are not verified by any third-party

As presented in Section 4.5.1 and discussed in Section 6.2.1, there were ambiguous terms in the CISQ specification, which could mean that many of the implementations of the CISQ measures in SQUIDS are incorrect from what was intended by the specification. The comparison of analysis results with Plösch, Schürz, and Körner (see Section 6.4) revealed that there are implemented CISQ measures which are excessively

different between SQuIDS and MUSE. However, no evidence has been found that the raw results from Plösch, Schürz, and Körner are verified by any third party. This means that if CISQ measures in MUSE are incorrect from the specification, the comparison of analysis results from SQuIDS and MUSE cannot contribute to the understanding of the correctness of either software.

6.6.2 External Validity

In order to make use of the results from this study, a few considerations have to be made. In both the analysis results and performance tests, only five existing software projects and only 16 out of 20 implemented CISQ measures were used.

6.6.2.1 Test Subjects Not Representative

The five software projects[6] were chosen because of their usage in the article which was used for comparison, but may not be representable for the average software project. As noted by Plösch, Schürz, and Körner (2015), the projects are all open-source, which may differ from "industrial strength projects". Testing more software projects, especially larger projects for industrial use, would be a great improvement to the tests performed in this project.

6.6.2.2 Ignored CISQ Measures

As stated in Section 5.3, only 16 out of the 20 implemented CISQ maintainability measures were used for gathering result data from analysis of the five open-source projects and the performance tests. Not providing data from the remaining four CISQ measures is a threat to the external validity if other research projects do, and wish to compare results. Hypothetically, if this study was to include them, it would either have to analyze software projects which the researcher has intricate knowledge of, or make the same compromises as Plösch, Schürz, and Körner's *Variant B* of their study (2015). In this variant, automatic detection of architectural layers is provided by an external tool called *Classycle* in order to operationalize CISQ measures M01-M03. However, according to Plösch, Schürz, and Körner, Classycle does unfortunately not produce the same description of the layering as the actual layering. A list of the standard Java JDK packages and classes which contain file or data operations is provided for CISQ measure M19, but other packages and classes of the same nature

[6] TV-Browser was ignored in comparing results (see Section 6.4).

are not considered. Performing tests with variant B would only provide less accurate results.

6.6.2.3 Reproducing Setup

Performance testing was performed on a single machine, as detailed in Table 3.4. Therefore, results may vary if another setup is used. Since other machines have not been used, the impact of the different attributes of a setup cannot be calculated. Therefore, if a comparison on performance is to be made with SQuIDS attempting to recreate the setup in this project as close as possible is recommended.

6.7 Chapter Summary

This chapter has examined the methods and relevant theory in the study, discussed the implementation of the CISQ specification, compared the results with Plösch, Schürz, and Körner (2015), as well as answered the research questions and discussing the validity of the results.

Chapter 7

Conclusions

This chapter starts with a brief summary of the thesis, before presenting the research contributions made. Finally, recommendations for further research and development is given.

7.1 Thesis Summary

This thesis has documented a research project using the Design Science Research framework, which resulted in the software artifact SQUIDS. The artifact was developed in order to examine how an Eclipse plug-in which automatically detects problems related to software maintainability, and visualize them to the user, in an acceptable time, can be developed.

The development process of SQUIDS is described by its requirements, scenarios, design and 17 iterations. Its correctness and performance were monitored in each iteration, based on making the software analyze a small project. By the end of the 16^{th} iteration, five existing open-source projects were analyzed by SQUIDS, and the data revealed incorrect results and slow performance, resulting in an additional iteration of improvements. The final results show acceptable performance and correctness, although improvement areas were identified. At the end of the thesis, the methods and theory used in the project are discussed, and the research questions are answered based on the test results.

7.2 Research Contribution

The DSR framework specifies in its fourth guideline (see Section 3.1.4) that the research must make a research contribution. This study contributes to the research

areas software engineering and software quality analysis, and specifically the automation of the maintainability characteristic in the CISQ specification, which is based on the ISO/IEC 25010 standard. Previous research exists in this area, although no other study was found which attempts to implement such a tool as a plug-in, while keeping its performance in mind[1].

The study has contributed the following results:

- A software artifact called SQuIDS: A plug-in for the Eclipse IDE which automatically analyzes Java source code and detects problems related to software maintainability, in accordance with the CISQ specification. It was found that analysis results from SQuIDS differed from another implementation of the same specification, with the same analysis subjects (the five open-source projects). This suggests that either or both implementations contain erroneously implemented metrics (CISQ measures).

- Proof that such a software may deliver results in an acceptable time. It is nevertheless shown that the type of analysis which is demanded by many of the metrics (CISQ measures) in the CISQ specification is time consuming, and that optimization and compromises have to be made to ensure satisfactory performance.

- An example and discussion of how software quality problems may be displayed to a user. It has been noted that the solution in the artifact may not be optimal, and that usability testing should be used to find a better solution.

- A method to compare and verify results between different implementations of the CISQ specification (see Section 3.2.3.3 and Section 6.4). By using the method for a selected set of metrics, it was found that the MUSE software failed to find some maintainability problems which were detected by SQuIDS (see Section 6.4.1.3). It was also found that MUSE counts inheritance levels differently than SQuIDS, on account to different interpretations of the CISQ specification (see Section 6.4.1.2).

In addition, the study provides a description of how the software artifact was designed and developed, so that other researchers and developers may use the information to software for similar purposes. SQuIDS is also available as an open-source project at https://github.com/larsac07/SQuIDS.

[1]Satrijandi and Widyani (2015) focus on performance, but does not implement the maintainability characteristic

7.3 Further Research and Development

During the project, some limitations and issues in the study were found. These provide opportunities for new research topics which may be of interest for further research.

7.3.1 Further Evaluation

The study was found to have three limitations which could be resolved by further research:

- Results not verified by third party
- Test subjects may not be representative
- Usability testing was not performed

7.3.1.1 Verification of Results

The correctness of the implemented metrics in the CISQ specification cannot be guaranteed, for three reasons:

- The results from SQuIDS have not been verified by a third party
- Comparison of results with the MUSE software created by Plösch, Schürz, and Körner (2015) showed different results
- The results from MUSE have not been verified by a third party

Therefore, a complete verification of the results of SQuIDS or MUSE, or a comparison with a software whose results are verified should be conducted. However, the detailed comparison of the results from selected CISQ measures in SQuIDS and MUSE found that MUSE fails to detect some maintainability problems which SQuIDS finds, reports duplicate problems and and makes different assumptions about the scope of inheritance levels (see Section 6.4). The comparison can also be replicated for other measures by using the same method, as described in Section 3.2.3.3.

7.3.1.2 Analyzing More Representative Subjects

The software projects which were analyzed (see Section 3.2.3.5) were all open-source, and there were only five of them[2] (see Section 6.6.2.1). In addition, four implemented

[2]Only four used to compare results with MUSE.

CISQ measures were ignored when analyzing the projects, because they required intimate knowledge of the projects. Analysis of a larger amount of projects, varying between open-source and proprietary software, where the knowledge required for the ignored CISQ measures is available, would provide a more complete and realistic result.

7.3.1.3 Usability Testing

Although the DSR framework does not require optimal solutions (Hevner et al., 2004, p. 89), the manner of displaying maintainability problems to users could be improved by performing usability tests on different solutions, such as the alternatives noted in Section 6.5.3.

7.3.2 Further Development of SQuIDS

SQuIDS could be improved as a software on four areas:

- Performance
- CISQ specification completeness
- Using the Visitor pattern
- Support for other programming languages

7.3.2.1 Improving performance

The average performance of SQuIDS could be improved by reducing the amount of files and CISQ measures for every analysis. By running a complete analysis only once, and keeping track of changes made to files, the files which have not been changed since the last analysis do not have to be parsed by JavaParser anew. In addition, many of the CISQ measures are only concerned with individual files, in contrast to others, where changes in one file may influence whether there is an issue in another file. Therefore, if each `Measure` is given an attribute (field) for its scope, CISQ measures may be filtered out if they do not need to reanalyze unchanged files, thus using less time and improving performance.

7.3.2.2 Completing the CISQ Specification

Where SQUIDS contains implementations of only one fourth of the CISQ specification (the maintainability characteristic), implementing CISQ measures for another, or the rest of the specification, would be a large improvement on SQUIDS's usage area.

7.3.2.3 Refactoring to the Visitor Pattern

As noted in Section 6.3.2, the Visitor pattern, which is both supported and used by the JavaParser library, could be used to simplify and familiarize the `Measure` class for other developers.

7.3.2.4 Support for Other Programming Languages

SQUIDS currently only supports analysis of Java source code (see Section 5.1.2). In order to support other languages, the `Measure` and `IssueFinder` classes (see Fig. 4.3.2) need to be refactored to not depend on JavaParser-specific classes, as the JavaParser library only supports Java source code. This means that a generic representation of ASTs must be used, for example with ANTLR. Some CISQ measures may need to be generalized, while others may need language-specific implementations. Additionally, if a hypothetical programming language cannot be represented as ASTs, it would not be recommended to extend SQUIDS for that language, as traversing ASTs is the central design of SQUIDS.

7.3.3 Query-Language for ASTs

It has been found that implementing source code analysis for Java using an AST analysis approach, though feasible, is at times complicated to express, especially when the provided AST has limited information (see Section 4.5.2). It would be interesting to see if an alternative approach could be developed, where a query-language for ASTs, or a language more suitable for the relations between nodes in an AST would be used to express the CISQ measures. An example query, as an implementation of CISQ measure M10, could look like the following: `<variable> <hasModifier> <public>`. Points of interest would be to see if such an implementation would be easier to both develop and maintain, or have a higher/lower performance.

7.3.4 Automated Quality Assessment

Within the research area of automated software quality analysis, there should be focus on developing more quality measures which can be automated. As Plösch, Schürz, and Körner (2015) points out, there are software quality models with a wider range of metrics. Additionally, a query-language for ASTs, as described in Section 7.3.3, could also be used to express the rules in a software quality model.

Bibliography

Bagheri, Ebrahim and Dragan Gasevic (2011). "Assessing the Maintainability of Software Product Line Feature Models using Structural Metrics". In: *Software Quality Journal* 19.3, 579–612. ISSN: 09639314. DOI: 10.1007/s11219-010-9127-2. URL: http://link.springer.com/article/10.1007/s11219-010-9127-2.

Bakota, T et al. (2012). "A cost model based on software maintainability". In: *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, 316–25. DOI: 10.1109/ICSM.2012.6405288. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6405288.

Bessin, Geoffrey (2004). *The business value of software quality*. URL: https://www.ibm.com/developerworks/rational/library/dec04/bessin/.

Bourque, Pierre and Richard E. Fairley (2014). *Guide to the software engineering body of knowledge (swebok)*. Ed. by Pierre Bourque and Richard E. Fairley. 3.0. IEEE Computer Society, 335. ISBN: 0769551661. DOI: 10.1234/12345678. URL: http://www.computer.org/web/swebok/v3http://www.mendeley.com/research/guide-software-engineering-body-knowledge-swebok/.

Broeckman, Bart and Edwin Notenboom (2003). *Testing Embedded Software*. 1st ed. Addison-Wesley, 348. ISBN: 0321159861.

Chen, Jing and Xuyan Liu (2009). "Software Maintainability Metrics Based on the Index System and Fuzzy Method". In: *2009 1st International Conference on Information Science and Engineering, ICISE 2009*, 5117–20. ISBN: 9780769538877. DOI: 10.1109/ICISE.2009.1073. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5454554.

Chidamber, Shyam R. and Chris F. Kemerer (1994). "A Metrics Suite for Object Oriented Design". In: *IEEE Transactions on Software Engineering* 20.6, 476–93. URL: http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=295895.

Chirilă, Ciprian-Bogdan and Vladimir Creţu (2012). "A Set of Java Metrics for Software Quality Tree Based on Static Code Analyzers". In: *Applied Computational Intelligence in Engineering and Information Technology*. Ed. by Radu-Emil Precup et al. 1st ed. Springer. Chap. 12, 147–62. URL: http://link.springer.com/content/pdf/10.1007/978-3-642-28305-5{_}12.pdf.

CISQ. *CISQ - Consortium for IT Software Quality*. URL: http://it-cisq.org/ (visited on 01/14/2016).

– (2012). *CISQ Specifications for Automated Quality Characteristic Measures*. URL: http://it-cisq.org/wp-content/uploads/2012/09/CISQ-Specification-for-Automated-Quality-Characteristic-Measures.pdf (visited on 11/27/2015).

Cohn, Mike (2006). *Agile Estimating and Planning*. Addison-Wesley, 308. URL: http://synchronit.com/downloads/books/AgileEstimatingandPlanning.pdf.

Coleman, Don et al. (1994). "Using Metrics to Evaluate Software System Maintainability". In: *Computer* 27.8, 44–49. ISSN: 0018-9162. DOI: 10.1109/2.303623. URL: http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=303623.

Cornell University (2005). *Strong versus weak typing*. URL: http://www.cs.cornell.edu/courses/cs1130/2012sp/1130selfpaced/module1/module1part4/strongtyping.html (visited on 02/11/2016).

Dahiya, S. Singh, J.K. Chhabra, and S. Kumar (2007). "Use of Genetic Algorithm for Software Maintainability Metrics' Conditioning". In: *15th International Conference on Advanced Computing and Communications (ADCOM 2007)*. Vol. 136119, 87–92. ISBN: 0-7695-3059-1. DOI: 10.1109/ADCOM.2007.69. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4425956.

Douziech, Philippe-Emmanuel (2016). *CISQ Question*. Email correspondence.

Fowler, Martin (2004). *UML Distilled*. 3rd ed. Addison-Wesley, 167. ISBN: 0-321-19368-7.

– (2015). *PresentationDomainDataLayering*. URL: http://martinfowler.com/bliki/PresentationDomainDataLayering.html (visited on 12/31/2015).

Goyvaerts, Jan (2015). *Regular-Expressions.info*. URL: http://www.regular-expressions.info/ (visited on 02/10/2016).

Grune, Dick (2012). *Modern Compiler Design*. New York, NY: Springer New York, 736. ISBN: 978-1-4614-4699-6. DOI: 10.1007/978-1-4614-4699-6. URL: http://www.academiaworld.net/wp-content/uploads/2015/08/Modern-Compiler-Design-2e.pdf.

Hevner, Alan R. et al. (2004). *Design Science in Information Systems Research*. URL: http://wise.vub.ac.be/thesis{_}info/design{_}science.pdf (visited on 02/29/2016).

IEEE Standards Association (2006). "Software Life Cycle Processes -Maintenance ISO/IEC 14764 IEEE Std 14764-2006". In: *ISO/IEC 14764:2006 (E) IEEE Std 14764-2006 Revision of IEEE Std 1219-1998)*, 0_1–46. DOI: 10.1109/IEEESTD.2006.235774.

– (2010). "Systems and software engineering — Vocabulary ISO/IEC/IEEE 24765:2010". In: *ISO/IEC/IEEE 24765:2010*, 1–418. DOI:

10 . 1109 / IEEESTD . 2010 . 5733835. URL: http://ieeexplore.ieee.org/servlet/opac?punumber=5733833.

Iso25000.com (2015). *ISO 25010*. URL: http://iso25000.com/index.php/en/iso-25000-standards/iso-25010?limit=3{\&}limitstart=0 (visited on 05/20/2016).

ISO/IEC (2011). *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. DOI: ISO / IECFDIS25010 : 2010(E). URL: http://www.iso.org/iso/catalogue{_}detail.htm?csnumber=35733 (visited on 03/04/2016).

Jones, Joel (2003). *Abstract Syntax Tree Implementation Idioms*. URL: http://www.hillside.net/plop/plop2003/Papers/Jones-ImplementingASTs.pdf (visited on 02/11/2016).

Kerievsky, Joshua (2004). *Refactoring to Patterns*. Pearson Higher Education, 367. ISBN: 0321213351.

Li, N, U Praphamontripong, and J Offutt (2009). "An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-Uses and Prime Path Coverage". In: *Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on*, 220–29. DOI: 10.1109/ICSTW.2009.30.

Lincke, Rüdiger, Jonas Lundberg, and Welf Löwe (2008). "Comparing Software Metrics Tools". In: *Proceedings of the 2008 international symposium on Software testing and analysis - ISSTA '08*, 131. URL: http://portal.acm.org/citation.cfm?doid=1390630.1390648.

Lucas, Jason (2006). *Thoughts on the Visual C++ Abstract Syntax Tree (AST) | Visual C++ Team Blog*. URL: https://blogs.msdn.microsoft.com/vcblog/2006/08/16/thoughts-on-the-visual-c-abstract-syntax-tree-ast/ (visited on 02/11/2016).

Martin, Robert (2009). *Clean code : a handbook of Agile software craftsmanship*. Vol. 1, 464. ISBN: 0132350882. URL: http://portal.acm.org/citation.cfm?id=1388398.

Maurer, Ward Douglas (1966). "A Theory of Computer Instructions". In: *J. ACM* 13.2, 226–35. ISSN: 0004-5411. DOI: 10.1145/321328.321334. URL: http://doi.acm.org/10.1145/321328.321334.

McCabe, T J (1976). "A Complexity Measure". In: *IEEE Transactions on Software Engineering* SE-2.4, 308–20. ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233837.

McCabe, T.J., D.R. Wallace, and A.H. Watson (1996). "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric". In: *NIST Special Publication* 500.235, 1–124. ISSN: 00831883. DOI: 800.638.6316. URL: http://www.mccabe.com/pdf/mccabe-nist235r.pdf.

Mitra, Tilak (2008). *Documenting software architecture, Part 3: Develop the architecture overview*. en. URL: http://www.ibm.com/developerworks/library/ar-archdoc3/.

MSDN (2009). *Chapter 5: Layered Application Guidelines*. URL: https://msdn.microsoft.com/en-us/library/ee658109.aspx (visited on 01/01/2016).

Muthanna, S et al. (2000). "A Maintainability Model for Industrial Software Systems using Design Level Metrics". In: *Seventh Working Conference on Reverse Engineering - Proceedings*, 248–56. ISBN: 0-7695-0881-2. DOI: 10.1109/WCRE.2000.891476. URL: http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=891476.

Nguyen, Vu et al. (2007). *A SLOC Counting Standard*. URL: http://sunset.usc.edu/TECHRPTS/2007/usc-csse-2007-737/usc-csse-2007-737.pdf (visited on 11/22/2015).

Nielsen, Jakob (1993). *Response Times: The 3 Important Limits*. URL: https://www.nngroup.com/articles/response-times-3-important-limits/ (visited on 05/27/2016).

Nystrom, Nathaniel, Michael R Clarkson, and Andrew C Myers (2003). "Compiler Construction: 12th International Conference, CC 2003 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003 Warsaw, Poland, April 7–11, 2003 Proceedings". In: ed. by Görel Hedin. Berlin, Heidelberg: Springer Berlin Heidelberg. Chap. Polyglot: 138–52. ISBN: 978-3-540-36579-2. DOI: 10.1007/3-540-36579-6_11. URL: http://dx.doi.org/10.1007/3-540-36579-6{_}11.

Object Management Group (2011). *Business Process Model and Notation*. URL: http://www.omg.org/spec/BPMN/2.0/PDF.

Oman, Paul and Jack Hagemeister (1992). "Metrics for Assessing a Software System's Maintainability". In: *Proceedings Conference on Software Maintenance 1992*. IEEE Comput. Soc. Press, 337–44. ISBN: 0-8186-2980-0. DOI: 10.1109/ICSM.1992.242525. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=242525.

Oracle (2015a). *About the Java Technology*. URL: http://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html (visited on 02/10/2016).

– (2015b). "Multiple Inheritance of State, Implementation, and Type". In: URL: https://docs.oracle.com/javase/tutorial/java/IandI/multipleinheritance.html.

– (2016). *The Java Virtual Machine Instruction Set*. URL: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html (visited on 02/22/2016).

Plösch, R., S. Schürz, and C. Körner (2015). "On the Validity of the IT-CISQ Quality Model for Automatic Measurement of Maintainability". In: *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*. IEEE Computer Society,

326 –334. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7273636http://ieeexplore.ieee.org/stamp/stamp.jsp?tp={\&}arnumber=7273636.

Pressman, Roger S. (2010). *Software Engineering: A Practitioner's Approach*. 7th ed. McGraw-Hill Higher Education, 895. ISBN: 978–0–07–337597–7. URL: http://nlp.chonbuk.ac.kr/SE/pressman07book.pdf.

Rashid, Muhammad and Bernard Pottier (2012). "Visitor-based application analysis methodology for early design space exploration". In: *Design Automation for Embedded Systems* 16.4, 319–38. ISSN: 1572-8080. DOI: 10.1007/s10617-013-9111-8. URL: http://dx.doi.org/10.1007/s10617-013-9111-8.

Rojas, José Miguel et al. (2015). "Search-Based Software Engineering: 7th International Symposium, SSBSE 2015, Bergamo, Italy, September 5-7, 2015, Proceedings". In: ed. by Márcio Barros and Yvan Labiche. Cham: Springer International Publishing. Chap. Combining, 93–108. ISBN: 978-3-319-22183-0. DOI: 10.1007/978-3-319-22183-0_7. URL: http://dx.doi.org/10.1007/978-3-319-22183-0{_}7.

Saraiva, Juliana, Sérgio Soares, and Fernando Castor (2013). "Towards a Catalog of Object-Oriented Software Maintainability Metrics". In: *International Workshop on Emerging Trends in Software Metrics, WETSoM*, 84–87. ISBN: 9781467363310. DOI: 10.1109/WETSoM.2013.6619342. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6619342.

Satrijandi, Nugroho and Yani Widyani (2015). *Efficiency Measurement of Java Android Code*. DOI: 10.1109/ICODSE.2014.7062696. URL: http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=7062696 (visited on 03/03/2016).

Schürz, Severin (2016). *Comparing results with On the Validity of the IT-CISQ Quality Model for Automatic Measurement of Maintainability*. Email correspondence.

Sen, Koushik, Darko Marinov, and Gul Agha (2005). "Cute". In: *ACM SIGSOFT Software Engineering Notes* 30.5, 263. ISSN: 01635948. DOI: 10.1145/1095430.1081750. URL: http://portal.acm.org/citation.cfm?doid=1095430.1081750.

Shen, Haihao, Sai Zhang, and Jianjun Zhao (2008). "An Empirical Study of Maintainability in Aspect-Oriented System Evolution Using Coupling Metrics". In: *2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering*, 233–36. DOI: 10.1109/TASE.2008.17. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4549910.

Sjøberg, Dag I K, Bente Anda, and Audris Mockus (2012). "Questioning Software Maintenance Metrics: A Comparative Case Study". In: *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '12. New York, NY, USA: ACM, 107–10. ISBN: 978-1-4503-1056-7. DOI:

10 . 1145 / 2372251 . 2372269. URL: http://doi.acm.org/10.1145/2372251.2372269.

Software Engineering Institute (2009). *Carnegie Mellon SEI and OMG Announce the Launch of CISQ—The Consortium for IT Software Quality (www.it-cisq.org)*. URL: http://www.sei.cmu.edu/newsitems/cisq.cfm (visited on 01/14/2016).

Sommerville, Ian (2011). *Software Engineering*. URL: https://acadndtechy.files.wordpress.com/2015/01/software{_}engineering{_}9th{_}edition{_}.pdf (visited on 02/15/2016).

The Eclipse Foundation (2016). *Eclipse desktop & web IDE*. URL: http://www.eclipse.org/ide/ (visited on 02/12/2016).

Vliet, Hans van (2007). *Software Engineering: Principles and Practice*. 3rd ed. Wiley, 560. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.128.2614{\&}rep=rep1{\&}type=pdf.

Wells, Don (1999a). *Iteration Planning*. URL: http://www.extremeprogramming.org/rules/iterationplanning.html (visited on 02/27/2016).

– (1999b). *Project Velocity*. URL: http://www.extremeprogramming.org/rules/velocity.html (visited on 02/28/2016).

– (1999c). *User Stories*. URL: http://www.extremeprogramming.org/rules/userstories.html (visited on 02/12/2016).

– (2000). *Test First*. URL: http://www.extremeprogramming.org/rules/testfirst.html (visited on 05/21/2016).

– (2009). *XP Values*. URL: http://www.extremeprogramming.org/values.html (visited on 05/23/2016).

Wikimedia Commons (2001). *XP-feedback*. URL: https://en.wikipedia.org/wiki/File:XP-feedback.gif (visited on 03/24/2016).

Xu, Baolin and Shiming Wan (2015). *The Design Strategy of Component Method in Three-Tier Architecture*. DOI: 10 . 1109 / ICISCE . 2015 . 116. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7120656 (visited on 01/01/2016).

Appendix A

User Stories

All user stories, except stories 1-3 and 7, are direct adaptations from the CISQ Maintainability Measures.

1. As a developer I want to be notified about empty catch-blocks so that I can improve exception handling

2. As a developer or future collaborator, I want the library to provide for AST analysis so that a measure can handle complex queries and improve speed and maintainability

3. As a developer or future collaborator I want the IssueFinder class and the Measure subclasses to be independent of the Eclipse framework, so that I may use it for other purposes

4. As a developer I want to be notified about layer-skipping calls so that I can improve code architecture

5. As a developer I want to be notified about too many horizontal layers so that I can improve code architecture

6. As a developer I want to be notified about functions that span layers so that I can improve code architecture

7. As a developer or future collaborator I want to be able to change, disable and add measures so that I can look for just certain issues or configure the software for other models

8. As a developer I want to be notified about files > 1000 LOC

9. As a developer I want to be notified about functions passing ≥ 7 parameters

10. As a developer I want to be notified about variables declared public, so that I can improve data access encapsulation

11. As a developer I want to be notified about functions that have a fan-out ≥ 10, so that I can reduce coupling between modules

12. As a developer I want to be notified about methods that are directly using fields from other classes so that I can improve data access encapsulation

13. As a developer I want to be notified about functions with $> 2\%$ commented out instructions so that I can reduce coupling between modules

14. As a developer I want to be notified about GO TOs, CONTINUE, and BREAK outside the switch

15. As a developer I want to be notified about hard coded literals except (-1, 0, 1, 2 or literals initializing static or constant variables)

16. As a developer I want to be notified about classes with ≥ 10 children

17. As a developer I want to be notified about files that contain 100+ consecutive duplicate tokens

18. As a developer I want to be notified about methods with ≥ 7 data or file operations

19. As a developer I want to be notified about functions with cyclomatic complexity $\geq$a language specific threshold (table to be inserted)

20. As a developer I want to be notified about unreachable functions

21. As a developer I want to be notified about classes with inheritance levels ≥ 7

22. As a developer I want measures to not discriminate between the terms method and function

23. As a developer I want to be notified about objects with coupling > 7

24. As a developer I want to be notified about cyclic calls between packages

25. As a developer I want to be notified about instances of indexes modified within its loop

26. As a developer I want to change settings and inputs via a GUI

27. As a developer I want a report of the total counts of rule violations per project

28. As a developer I want a percentage-done indicator of the analysis, so that if it indicates a long time, I may perform other tasks in the meantime

Appendix B

CISQ Measure Implementation Differences

M04 - It seems there are differences in definitions of tokens

M05 - SQuIDS does not detect method- and constructor-calls correctly: does not detect classes of objects produced by method-calls

M06 - While SQuIDS considers scope of inheritance as local, it seems MUSE follows the trail through imported libraries as well

M07 - TVBrowser the only anomaly. Different versions of TVBrowser analyzed

M09 - SQuIDS does not detect classes of objects produced by method-calls, and does not consider direct access from anonymous classes as a violation

M10 - TVBrowser the only anomaly. Different versions of TVBrowser analyzed

M11 - SQuIDS counts throwing of exceptions as well as method calls, object creations (`new Object(...);`) and explicit constructor invocations (`this()` / `super()`)

M12 - SQuIDS does not detect method- and constructor-calls correctly: does not detect classes of objects produced by method-calls

M13 - Cyclic calls between packages are reported differently

M14 - Commented out instructions analyzed differently

M15 - LOC calculated differently

M16 - MUSE does not consider modification of variables unless it is directly modifying

the value as it accesses the array, e.g. finds `array[i + 1];`, but does not find `i += 1; array[i];`

M17 - Jump statements may be defined differently are reported per jump statement in SQUIDS, and per control-flow block (if, for, while, etc.) in MUSE

M18 - SQUIDS and MUSE consider control flow statements differently

M20 - TVBrowser the only anomaly. Different versions of TVBrowser analyzed

M21 - MUSE does not consider empty strings as hard coded literals

Appendix C

Measure Class Diagram

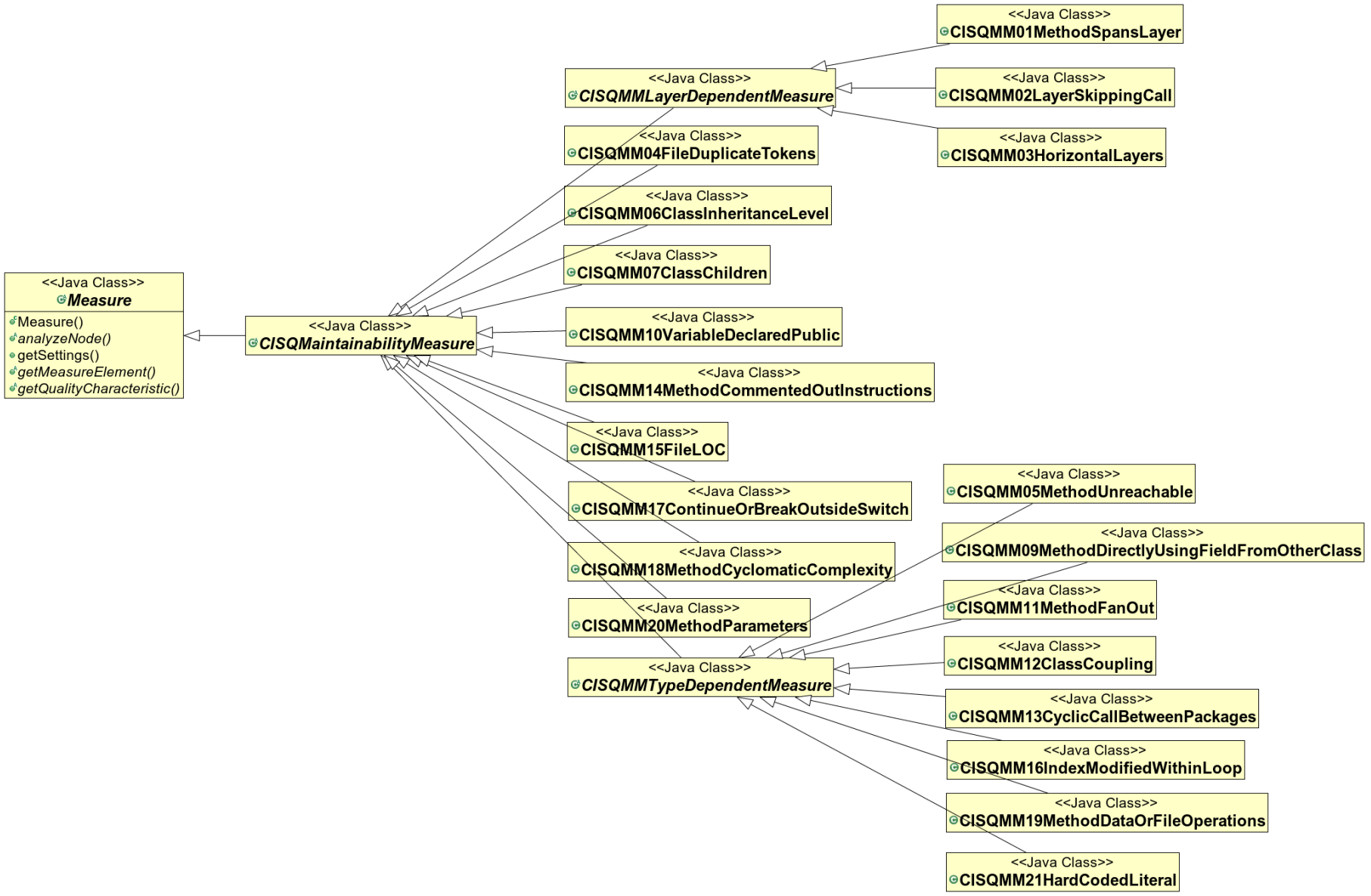

FIGURE C.1: Hierarchy of the `Measure` classes

Appendix D

Performance Test Result Tables

TABLE D.1: SQuIDS performance analyzing Checkstyle 4.4.

Measure	Time used (s)	Time used per 10 files (s)
M04	3.771	0.139
M05	0.201	0.007
M06	0.003	0.000
M07	1.005	0.037
M09	0.063	0.002
M10	0.004	0.000
M11	0.126	0.005
M12	0.449	0.017
M13	0.033	0.001
M14	0.067	0.002
M15	0.102	0.004
M16	0.280	0.010
M17	0.045	0.002
M18	0.133	0.005
M20	0.013	0.000
M21	0.503	0.018
Analysis time (measures total)	6.798	0.250
Parsing time	0.321	0.012
Marking time	2.843	0.105
Other operations	1.146	0.042
Total time	11.108	0.408

TABLE D.2: SQUIDS performance analyzing JabRef 2.3.1.

Measure	Time used (s)	Time used per 10 files (s)
M04	12.875	0.306
M05	0.598	0.014
M06	0.034	0.001
M07	6.720	0.160
M09	0.564	0.013
M10	0.202	0.005
M11	1.082	0.026
M12	17.245	0.410
M13	0.450	0.011
M14	0.510	0.012
M15	0.358	0.009
M16	0.884	0.021
M17	0.193	0.005
M18	4.247	0.101
M20	0.017	0.000
M21	13.496	0.321
Analysis time (measures total)	59.475	1.413
Parsing time	0.637	0.015
Marking time	23.007	0.546
Other operations	0.970	0.023
Total time	84.089	1.997

TABLE D.3: SQUIDS performance analyzing Log4j 1.2.15.

Measure	Time used (s)	Time used per 10 files (s)
M04	2.463	0.147
M05	0.137	0.008
M06	0.002	0.000
M07	0.430	0.026
M09	0.037	0.002
M10	0.002	0.000
M11	0.178	0.011
M12	0.047	0.003
M13	0.095	0.006
M14	0.070	0.004
M15	0.079	0.005
M16	0.190	0.011
M17	0.014	0.001
M18	0.083	0.005
M20	0.003	0.000
M21	1.199	0.071
Analysis time (measures total)	5.029	0.299
Parsing time	0.213	0.013
Marking time	3.578	0.213
Other operations	0.189	0.011
Total time	9.009	0.536

TABLE D.4: SQuIDS performance analyzing RSSOwl 1.2.4.

Measure	Time used (s)	Time used per 10 files (s)
M04	20.800	1.035
M05	0.445	0.022
M06	0.019	0.001
M07	1.733	0.086
M09	2.271	0.113
M10	0.142	0.007
M11	2.227	0.111
M12	0.278	0.014
M13	1.360	0.068
M14	0.172	0.009
M15	0.241	0.012
M16	0.898	0.045
M17	0.069	0.003
M18	1.668	0.083
M20	0.026	0.001
M21	59.755	2.973
Analysis time (measures total)	92.104	4.582
Parsing time	0.532	0.026
Marking time	95.195	4.736
Other operations	1.008	0.050
Total time	188.839	9.395

TABLE D.5: SQuIDS performance analyzing TV-Browser 2.2.6.

Measure	Time used (s)	Time used per 10 files (s)
M04	13.466	0.206
M05	0.946	0.014
M06	0.073	0.001
M07	9.095	0.139
M09	0.375	0.006
M10	0.045	0.001
M11	1.133	0.017
M12	1.123	0.017
M13	1.823	0.028
M14	0.267	0.004
M15	0.357	0.005
M16	1.328	0.020
M17	0.110	0.002
M18	3.981	0.061
M20	0.056	0.001
M21	8.791	0.135
Analysis time (measures total)	42.969	0.658
Parsing time	2.071	0.032
Marking time	53.890	0.825
Other operations	4.251	0.065
Total time	103.181	1.580

Appendix E

Comparisons of Problem Counts

TABLE E.1: Comparison of problem counts in Checkstyle 4.4.

CISQ Measure	SQUIDS count	MUSE count	Difference
M04	3	0	3
M05	113	8	105
M06	0	0	0
M07	3	3	0
M09	0	5	5
M10	0	0	0
M11	148	107	41
M12	110	90	20
M13	0	0	0
M14	5	18	13
M15	0	1	1
M16	46	6	40
M17	56	47	9
M18	59	20	39
M20	3	3	0
M21	1,018	1,029	11
Total	1,564	1,337	227

TABLE E.2: Comparison of problem counts in JabRef 2.3.1.

CISQ Measure	SQUIDS count	MUSE count	Difference
M04	10	0	10
M05	134	8	126
M06	0	12	12
M07	7	7	0
M09	336	708	372
M10	192	192	0
M11	808	594	214
M12	204	407	203
M13	767	70	697
M14	274	525	251
M15	10	14	4
M16	75	7	68
M17	188	123	65
M18	215	133	82
M20	11	11	0
M21	12,046	12,410	364
Total	15,277	15,221	56

TABLE E.3: Comparison of problem counts in Log4j 1.2.15.

CISQ Measure	SQUIDS count	MUSE count	Difference
M04	2	0	2
M05	68	8	60
M06	0	4	4
M07	1	1	0
M09	17	53	36
M10	4	4	0
M11	162	101	61
M12	24	83	59
M13	104	16	88
M14	29	48	19
M15	1	3	2
M16	7	2	5
M17	21	21	0
M18	23	13	10
M20	4	4	0
M21	1,578	1,587	9
Total	2,045	1,948	97

TABLE E.4: Comparison of problem counts in RSSOwl 1.2.4.

CISQ Measure	SQUIDS count	MUSE count	Difference
M04	4	0	4
M05	82	23	59
M06	0	1	1
M07	3	3	0
M09	540	946	406
M10	175	175	0
M11	524	348	176
M12	53	186	133
M13	1,329	97	1,232
M14	5	9	4
M15	3	13	10
M16	3	1	2
M17	53	50	3
M18	114	71	43
M20	4	4	0
M21	34,432	34,536	104
Total	37,324	36,463	861

TABLE E.5: Comparison of problem counts in TV-Browser 2.2.6.

CISQ Measure	SQuIDS count	MUSE count	Difference
M04	13	0	13
M05	251	12	239
M06	1	2	1
M07	3	2	1
M09	45	65	20
M10	38	35	3
M11	1,065	702	363
M12	77	438	361
M13	805	364	441
M14	56	59	3
M15	2	4	2
M16	94	3	91
M17	110	89	21
M18	178	91	87
M20	28	29	1
M21	11,422	12,957	1,535
Total	14,188	14,852	664

TABLE E.6: Comparison of total numbers of problems found per CISQ measure between SQuIDS and MUSE (Schürz, 2016), with the TV-Browser project.

CISQ Measure	SQuIDS count	MUSE count	Difference
M04	32	0	32
M05	648	59	589
M06	1	19	18
M07	17	16	1
M09	938	1,777	839
M10	409	406	3
M11	2,707	1,852	855
M12	468	1,204	736
M13	3,005	547	2,458
M14	369	659	290
M15	16	35	19
M16	225	19	206
M17	428	330	98
M18	589	328	261
M20	50	51	1
M21	60,496	62,519	2,023
Total	70,398	69,821	577

Appendix F

Manual Inspection of Results from MUSE and SQUIDS

F.1 M06

TABLE F.1: Notes from manual inspection of M06 findings. All problems were reported by MUSE.

File path	Notes
`JABREF_ROOT/src/java/net/sf/jabref/FileHistory.java:9`	extends Java Swing class `JMenu`. w/Object: 7, w.o/Object: 6
`JABREF_ROOT/src/java/net/sf/jabref/HelpContent.java:53`	extends Java Swing class `JTextPane`. w/Object: 7, w.o/Object: 6
`JABREF_ROOT/src/java/net/sf/jabref/SearchManager2.java:44`	extends `SidePaneComponent` which extends JGoodies class `SimpleInternalFrame`. w/Object: 7, w.o/Object: 6
`JABREF_ROOT/src/java/net/sf/jabref/collab/FileUpdatePanel.java:11`	extends `SidePaneComponent` which extends JGoodies class `SimpleInternalFrame`. w/Object: 7, w.o/Object: 6
`JABREF_ROOT/src/java/net/sf/jabref/groups/GroupSelector.java:43`	extends `SidePaneComponent` which extends JGoodies class `SimpleInternalFrame`. w/Object: 7, w.o/Object: 6
`JABREF_ROOT/src/java/net/sf/jabref/gui/MainTable.java:479`	extends GeneralRenderer which extends Java Swing class `DefautTableCellRenderer`. w/Object: 7, w.o/Object: 6

Continued on next page

Table F.1 – continued from previous page

File path	Notes
JABREF_ROOT/src/java/net/sf/jabref/gui/MainTable.java:494	extends GeneralRenderer which extends Java Swing class DefautTableCellRenderer. w/Object: 7, w.o/Object: 6
JABREF_ROOT/src/java/net/sf/jabref/imports/CiteSeerFetcher.java:46	extends SidePaneComponent which extends JGoodies class SimpleInternalFrame. w/Object: 7, w.o/Object: 6
JABREF_ROOT/src/java/net/sf/jabref/imports/CiteSeerFetcherPanel.java:38	extends SidePaneComponent which extends JGoodies class SimpleInternalFrame. w/Object: 7, w.o/Object: 6
JABREF_ROOT/src/java/net/sf/jabref/imports/GeneralFetcher.java:24	extends SidePaneComponent which extends JGoodies class SimpleInternalFrame. w/Object: 7, w.o/Object: 6
JABREF_ROOT/src/java/net/sf/jabref/imports/MedlineFetcher.java:32	extends SidePaneComponent which extends JGoodies class SimpleInternalFrame. w/Object: 7, w.o/Object: 6
JABREF_ROOT/src/java/net/sf/jabref/util/CaseChangeMenu.java:31	extends Java Swing class JMenu. w/Object: 7, w.o/Object: 6
RSSOWL_ROOT/src/java/net/sourceforge/rssowl/controller/forms/Hyperlink.java:51	extends AbstractHyperlink which extends Eclipse SWT class Canvas. w/Object: 7, w.o/Object: 6
LOG4J_ROOT/src/main/java/org/apache/log4j/lf5/viewer/LogFactor5ErrorDialog.java:33	extends LogFactor5Dialog which extends Java Swing class JDialog. w/Object: 7, w.o/Object: 6
LOG4J_ROOT/src/main/java/org/apache/log4j/lf5/viewer/LogFactor5InputDialog.java:38	extends LogFactor5Dialog which extends Java Swing class JDialog. w/Object: 7, w.o/Object: 6
LOG4J_ROOT/src/main/java/org/apache/log4j/lf5/viewer/LogFactor5LoadingDialog.java:31	extends LogFactor5Dialog which extends Java Swing class JDialog. w/Object: 7, w.o/Object: 6

Continued on next page

Table F.1 – continued from previous page

File path	Notes
`LOG4J_ROOT/src/main/java/org/apache/log4j/lf5/viewer/categoryexplorer/CategoryNodeEditorRenderer.java:31`	extends `CategoryNodeRenderer` which extends Java Swing class `DefaultTreeCellRenderer`. w/Object: 7, w.o/Object: 6

F.2 M17

TABLE F.2: Notes from manual inspection of M17 findings. All problems were reported by SQuIDS.

File path	Notes
`CHECKSTYLE_ROOT/src/checkstyle/com/puppycrawl/tools/checkstyle/api/ScopeUtils.java:125`	`break` inside `else if` inside `for`
`CHECKSTYLE_ROOT/src/checkstyle/com/puppycrawl/tools/checkstyle/api/ScopeUtils.java:129`	`break` inside `else if` inside `for`
`CHECKSTYLE_ROOT/src/checkstyle/com/puppycrawl/tools/checkstyle/api/ScopeUtils.java:160`	`break` inside `else if` inside `for`
`CHECKSTYLE_ROOT/src/checkstyle/com/puppycrawl/tools/checkstyle/api/ScopeUtils.java:164`	`break` inside `else if` inside `for`
`CHECKSTYLE_ROOT/src/checkstyle/com/puppycrawl/tools/checkstyle/api/ScopeUtils.java:208`	`break` inside `else if` inside `for`
`CHECKSTYLE_ROOT/src/checkstyle/com/puppycrawl/tools/checkstyle/api/ScopeUtils.java:212`	`break` inside `else if` inside `for`
`CHECKSTYLE_ROOT/src/checkstyle/com/puppycrawl/tools/checkstyle/api/TokenTypes.java:3354`	`continue` inside `if` inside `for` inside static block
`CHECKSTYLE_ROOT/src/checkstyle/com/puppycrawl/tools/checkstyle/checks/coding/FinalLocalVariableCheck.java:112`	`break` inside `if` inside `switch`

Continued on next page

Table F.2 – continued from previous page

File path	Notes
CHECKSTYLE_ROOT/src/checkstyle/com/puppycrawl/tools/checkstyle/checks/duplicates/StrictDuplicateCodeCheck.java:422	`continue` inside `if` inside `for` (final class)
CHECKSTYLE_ROOT/src/checkstyle/com/puppycrawl/tools/checkstyle/checks/duplicates/StrictDuplicateCodeCheck.java:429	`continue` inside `if` inside `if` inside `for` (final class)
JABREF_ROOT/src/java/net/sf/ext/BrowserLauncher.java:423	`continue` inside `if` inside try inside `for` inside `switch`
JABREF_ROOT/src/java/net/sf/jabref/AuthorList.java:361	`break` with label inside `if` inside `switch` with same label
JABREF_ROOT/src/java/net/sf/jabref/AuthorList.java:363	`break` with label inside `if` inside `switch` with same label
JABREF_ROOT/src/java/net/sf/jabref/AuthorList.java:367	`break` with label inside `if` inside `if` inside `switch` with same label
JABREF_ROOT/src/java/net/sf/jabref/AuthorList.java:371	`break` with label inside `if` inside `switch` with same label
JABREF_ROOT/src/java/net/sf/jabref/BasePanel.java:745	`break` with label inside `if` inside `for` with same label (anonymous inner class)
JABREF_ROOT/src/java/net/sf/jabref/BasePanel.java:932	`break` inside `if` inside `for` (anonymous inner class)
JABREF_ROOT/src/java/net/sf/jabref/ContentSelectorDialog2.java:263	`continue` with label inside `if` inside `for` with same label
JABREF_ROOT/src/java/net/sf/jabref/GeneralTab.java:165	`break` with label inside `if` inside `for` with same label
JABREF_ROOT/src/java/net/sf/jabref/JabRef.java:674	`continue` with label inside `if` inside `for` inside `for` with same label
JABREF_ROOT/src/java/net/sf/jabref/JabRefFrame.java:1993	`break` with label inside `if` inside `if` inside `for` with same label
JABREF_ROOT/src/java/net/sf/jabref/JabRefFrame.java:1995	`break` with label inside `if` inside `for` with same label
JABREF_ROOT/src/java/net/sf/jabref/Util.java:1316	`continue` inside `if` inside `for`

Continued on next page

Table F.2 – continued from previous page

File path	Notes
JABREF_ROOT/src/java/net/sf/jabref/Util.java:911	`continue` with label inside `if` inside `if` inside `if` inside `for` inside `for` with same label
JABREF_ROOT/src/java/net/sf/jabref/Util.java:924	`continue` with label inside `if` inside `if` inside `for` inside `if` inside `foreach` with same label
JABREF_ROOT/src/java/net/sf/jabref/bst/BibtexNameFormatter.java:100	`continue` inside `if` with sibling `switch` inside `if` inside `while`
JABREF_ROOT/src/java/net/sf/jabref/bst/BstLexer.java:854	`break` with label inside `if` inside `switch` inside `do while` with same label
JABREF_ROOT/src/java/net/sf/jabref/bst/BstLexer.java:944	`break` with label inside `if` inside `switch` inside `do while` with same label
JABREF_ROOT/src/java/net/sf/jabref/bst/BstParser.java:103	`break` with label inside `if` inside `switch` inside `do while` with same label
JABREF_ROOT/src/java/net/sf/jabref/bst/BstParser.java:1079	`break` with label inside `if` inside `switch` inside `do while` with same label
JABREF_ROOT/src/java/net/sf/jabref/bst/BstParser.java:660	`break` with label inside `if` inside `switch` inside `do while` with same label
JABREF_ROOT/src/java/net/sf/jabref/bst/VM.java:838	`break` inside `if` inside `do while`
JABREF_ROOT/src/java/net/sf/jabref/collab/ChangeScanner.java:139	`continue` with label inside `if` inside `for` with same label
JABREF_ROOT/src/java/net/sf/jabref/collab/ChangeScanner.java:152	`continue` with label inside `if` inside `if` inside `for` with same label
JABREF_ROOT/src/java/net/sf/jabref/collab/ChangeScanner.java:275	`break` with label inside `if` inside `for` with same label
JABREF_ROOT/src/java/net/sf/jabref/collab/ChangeScanner.java:329	`continue` with label inside `if` inside `if` inside `for` inside `for` with same label

Continued on next page

Table F.2 – continued from previous page

File path	Notes
JABREF_ROOT/src/java/net/sf/jabref/collab/ChangeScanner.java:368	`break` with label inside `if` inside `for` with same label
JABREF_ROOT/src/java/net/sf/jabref/export/layout/LayoutHelper.java:524	`break` inside `while`
JABREF_ROOT/src/java/net/sf/jabref/export/layout/format/HTMLChars.java:58	`break` with label inside `if` inside `if` with same label
JABREF_ROOT/src/java/net/sf/jabref/export/layout/format/RTFChars.java:57	`break` with label inside `if` inside `if` with same label
JABREF_ROOT/src/java/net/sf/jabref/export/layout/format/RTFChars.java:89	`break` with label inside `if` inside `if` with same label
JABREF_ROOT/src/java/net/sf/jabref/export/layout/format/WrapFileLinks.java:118	`break` inside `if` inside `switch` inside `foreach`
JABREF_ROOT/src/java/net/sf/jabref/export/layout/format/WrapFileLinks.java:151	`break` inside `if` inside `switch` inside `foreach`
JABREF_ROOT/src/java/net/sf/jabref/external/FileLinksUpgradeWarning.java:198	`break` with label inside `if` inside `for` inside `for` with same label
JABREF_ROOT/src/java/net/sf/jabref/groups/GroupSelector.java:1289	`break` inside `for`
JABREF_ROOT/src/java/net/sf/jabref/gui/EntryCustomizationDialog2.java:201	`continue` inside `if` inside `for`
JABREF_ROOT/src/java/net/sf/jabref/gui/ImportInspectionDialog.java:538	`break` inside `if` inside `for`
JABREF_ROOT/src/java/net/sf/jabref/imports/BiblioscapeImporter.java:256	`continue` inside `if` inside `if` inside `while`

Continued on next page

Table F.2 – continued from previous page

File path	Notes
JABREF_ROOT/src/java/net/sf/jabref/imports/BibtexParser.java:190	`break` inside `while`
JABREF_ROOT/src/java/net/sf/jabref/imports/BibtexParser.java:496	`break` inside `if` inside `while`
JABREF_ROOT/src/java/net/sf/jabref/imports/CopacImporter.java:118	`continue` inside `if` inside `for`
JABREF_ROOT/src/java/net/sf/jabref/imports/CsaImporter.java:244	`continue` inside `if` inside `while`
JABREF_ROOT/src/java/net/sf/jabref/imports/IsiImporter.java:273	`continue` with label inside `if` inside `if` inside `for` with same label
JABREF_ROOT/src/java/net/sf/jabref/imports/IsiImporter.java:283	`continue` inside `if` inside `if` inside `for` with a label
JABREF_ROOT/src/java/net/sf/jabref/imports/IsiImporter.java:293	`continue` with label inside `if` inside `if` inside `for` with same label
JABREF_ROOT/src/java/net/sf/jabref/imports/JstorImporter.java:56	`break` with label inside `if` inside `while` with same label
JABREF_ROOT/src/java/net/sf/jabref/imports/MedlineFetcher.java:312	`break` inside `if` inside `for`
JABREF_ROOT/src/java/net/sf/jabref/imports/MedlineFetcher.java:323	`break` inside `if` inside `for`
JABREF_ROOT/src/java/net/sf/jabref/imports/OpenDatabaseAction.java:304	`break` with label inside `if` inside `if` with same label
JABREF_ROOT/src/java/net/sf/jabref/imports/OpenDatabaseAction.java:308	`break` with label inside `if` inside `for` inside `if` with same label

Continued on next page

Table F.2 – continued from previous page

File path	Notes
JABREF_ROOT/src/java/net/sf/jabref/imports/SilverPlatterImporter.java:80	`continue` with label inside `if` inside `for` with same label
JABREF_ROOT/src/java/net/sf/jabref/imports/SilverPlatterImporter.java:85	`continue` with label inside `if` inside `for` with same label
JABREF_ROOT/src/java/net/sf/jabref/imports/SixpackImporter.java:96	`continue` with label inside `if` inside `while` with same label
JABREF_ROOT/src/java/net/sf/jabref/imports/TextAnalyzer.java:45	`continue` with label inside `if` inside `for` with same label
JABREF_ROOT/src/java/net/sf/jabref/imports/TextAnalyzer.java:54	`break` with label inside `if` inside `if` inside `if` inside `for` with same label
JABREF_ROOT/src/java/net/sf/jabref/imports/TextAnalyzer.java:89	`break` with label inside `if` inside `for` with same label
JABREF_ROOT/src/java/net/sf/jabref/labelPattern/LabelPatternUtil.java:476	`continue` with label inside `if` inside inside `while` with same label
JABREF_ROOT/src/java/net/sf/jabref/labelPattern/LabelPatternUtil.java:479	`continue` with label inside `if` inside `for` inside `while` with same label
JABREF_ROOT/src/java/net/sf/jabref/search/SearchExpressionLexer.java:112	`continue` with label inside `if` with sibling `switch` inside `for` with same label
JABREF_ROOT/src/java/net/sf/jabref/search/SearchExpressionLexer.java:256	`break` with label inside `if` inside `do while` with same label
JABREF_ROOT/src/java/net/sf/jabref/search/SearchExpressionLexer.java:300	`break` with label inside `if` inside `do while` with same label
JABREF_ROOT/src/java/net/sf/jabref/search/SearchExpressionTreeParser.java:189	`continue` inside `if` with sibling `switch` inside `for`

Continued on next page

Table F.2 – continued from previous page

File path	Notes
JABREF_ROOT/src/java/net/sf/jabref/util/XMPUtil.java:1210	`break` inside `if` inside `switch`
JABREF_ROOT/src/java/net/sf/jabref/util/XMPUtil.java:1228	`break` inside `if` inside `switch`
JABREF_ROOT/src/java/net/sf/jabref/util/XMPUtil.java:1237	`break` inside `if` inside `switch`
JABREF_ROOT/src/java/net/sf/jabref/util/XMPUtil.java:696	`continue` inside `if` inside `for`
JABREF_ROOT/src/java/net/sf/jabref/util/XMPUtil.java:715	`continue` inside `if` inside `for`
JABREF_ROOT/src/java/net/sf/jabref/util/XMPUtil.java:734	`continue` inside `if` inside `for`
JABREF_ROOT/src/java/net/sf/jabref/util/XMPUtil.java:762	`continue` inside `if` inside `for`
JABREF_ROOT/src/java/net/sf/jabref/util/XMPUtil.java:824	`continue` inside `if` inside `for`
RSSOWL_ROOT/src/java/net/sourceforge/rssowl/controller/RSSOwlLoader.java:316	`break` inside `if` inside `for`
RSSOWL_ROOT/src/java/net/sourceforge/rssowl/controller/dialog/SelectCategoryDialog.java:522	`break` inside `if` inside `for`
RSSOWL_ROOT/src/java/net/sourceforge/rssowl/controller/statusline/StatusLineAnimator.java:86	`break` inside `if` inside `while`
RSSOWL_ROOT/src/java/net/sourceforge/rssowl/controller/thread/FeedDiscoveryManager.java:281	`break` with label inside catch inside `while` inside `while` with same label
RSSOWL_ROOT/src/java/net/sourceforge/rssowl/controller/thread/FeedQueueLoader.java:116	`break` with label inside `if` inside `while` with same label
RSSOWL_ROOT/src/java/net/sourceforge/rssowl/controller/thread/FeedQueueLoader.java:95	`break` with label inside `if` inside `while` with same label

Continued on next page

Table F.2 – continued from previous page

File path	Notes
`RSSOWL_ROOT/src/java/net/sourceforge/rssowl/controller/thread/FeedSearchManager.java:312`	`break` with label inside catch inside `while` inside `while` with same label
`RSSOWL_ROOT/src/java/net/sourceforge/rssowl/controller/tray/SystemTrayAlert.java:396`	`break` inside catch inside `while`
`RSSOWL_ROOT/src/java/net/sourceforge/rssowl/util/shop/XMLShop.java:451`	`break` inside `if` inside `while`
`LOG4J_ROOT/src/main/java/org/apache/log4j/Dispatcher.java:101`	`break` inside catch inside synchronized(bf) inside `while`
`LOG4J_ROOT/src/main/java/org/apache/log4j/config/PropertyGetter.java:71`	`continue` inside `if` inside `for`
`LOG4J_ROOT/src/main/java/org/apache/log4j/config/PropertySetter.java:128`	`continue` inside `if` inside `for`
`LOG4J_ROOT/src/main/java/org/apache/log4j/net/SocketAppender.java:404`	`break` inside synchronized(this) inside `while`