

DEPARTMENT OF INFORMATICS  
PROGRAMMING THEORY

Master Thesis

---

# Novice Difficulties with Language Constructs

---

*Alexander Hoem Rosbach*

July 2013



## Abstract

Programming is a difficult skill to learn, and programming courses have high dropout rates. In this thesis we study the problems that students have during their first introductory programming course at The University of Bergen. We inspect the solutions that they submit for the given assignments, and look at the frequency of the different kinds of mistakes in their work.

We present a problem taxonomy that we use to classify the mistakes found to be the most common, and conclude that a significant part of the problems are observable misconceptions. We introduce a web-based tool, Jarvis, that we have developed to aid the students with these kinds of problems.

Based on the experience and knowledge gained during this work we present a proposal of a *grading by annotation* scheme. This scheme is specifically designed to increase the quality of the feedback given to students on their submitted work and provide valuable feedback to the teachers regarding the problems that their students have.



## **Acknowledgements**

Foremost, I want to thank my supervisor Anya Helene Bagge for providing the initial inspiration for the thesis, and contribution of ideas, support and advice during my work.

I would also like to thank May-Lill Bagge for her reviews and feedback of my work. Her non computer science perspective and input was a great help.



# Contents

<b>Contents</b>	<b>i</b>
<b>List of Figures</b>	<b>v</b>
<b>List of Tables</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background and Motivation . . . . .	1
1.2 Research Questions . . . . .	2
1.3 Related Work . . . . .	3
1.4 Thesis Structure and Outline . . . . .	3
1.4.1 Outline . . . . .	4
<b>2 Learning Programming</b>	<b>7</b>
2.1 Programming Skills . . . . .	8
2.2 The Challenges . . . . .	9
2.3 Student Problem Taxonomy . . . . .	11
2.3.1 Concept-based problems . . . . .	12
2.3.2 Knowledge-based problems . . . . .	13
2.4 INF100 Course . . . . .	14
<b>3 Study Design and Problem Taxonomy</b>	<b>17</b>
3.1 Study Description . . . . .	17
3.2 Data Set . . . . .	18
3.3 Problem Taxonomy – Most common problems . . . . .	20
3.3.1 $\begin{smallmatrix} \text{WRNG} \\ \swarrow \\ \text{COND} \end{smallmatrix}$ – Wrong condition . . . . .	20
3.3.2 $\begin{smallmatrix} \text{BDRY} \\ \swarrow \\ \text{COND} \end{smallmatrix}$ – Boundary case condition . . . . .	21
3.3.3 $\begin{smallmatrix} \text{LOOP} \\ \swarrow \\ \text{CREATE} \end{smallmatrix}$ – Loop instantiation problem . . . . .	22
3.3.4 $\begin{smallmatrix} \text{CMLX} \\ \swarrow \\ \text{CTRLS} \end{smallmatrix}$ – Unnecessarily complicated . . . . .	24
3.3.5 $\begin{smallmatrix} \text{NO} \\ \swarrow \\ \text{CALL} \end{smallmatrix}$ – Missing method call(s) . . . . .	25
3.3.6 $\begin{smallmatrix} \text{WRNG} \\ \swarrow \\ \text{GRPNG} \end{smallmatrix}$ – Incorrect grouping . . . . .	26

3.3.7	$\not\prec_{\text{ARIT}}^{\text{BAD}}$ – Erroneous arithmetic . . . . .	27
3.3.8	$\not\prec_{\text{ACCUM}}^{\text{BOOL}}$ – Accumulate boolean . . . . .	28
<b>4</b>	<b>Result Analysis</b>	<b>31</b>
4.1	Lazy students? . . . . .	31
4.2	Most common problems . . . . .	32
4.3	Parameter passing and references . . . . .	34
4.4	Conditions . . . . .	35
4.5	Loop constructs . . . . .	36
4.6	Method calls . . . . .	37
4.7	Concept-based problems . . . . .	37
4.8	Conclusion . . . . .	39
<b>5</b>	<b>Tool Assistance – Jarvis</b>	<b>41</b>
5.1	Requirements . . . . .	41
5.2	Implementation . . . . .	43
5.2.1	Java evaluation . . . . .	44
5.2.2	Abstract syntax tree . . . . .	45
5.3	Tool Design . . . . .	45
5.3.1	Design concepts . . . . .	46
5.3.2	Evaluation . . . . .	49
5.4	Visualisation Examples . . . . .	51
5.4.1	Step by step evaluation . . . . .	51
5.4.2	Nested scopes . . . . .	53
5.4.3	Parameter passing . . . . .	54
5.5	Related Work . . . . .	54
5.6	Conclusion . . . . .	56
<b>6</b>	<b>Proposal: Grading by Annotation</b>	<b>57</b>
6.1	Annotation Syntax . . . . .	58
6.2	Grading . . . . .	59
6.2.1	File annotations . . . . .	59
6.2.2	Meta annotations . . . . .	60
6.2.3	Problem annotations . . . . .	60
6.3	Collecting the Data . . . . .	63
<b>7</b>	<b>Conclusion</b>	<b>67</b>
7.1	Status Summary . . . . .	67
7.2	Future Work . . . . .	68
7.2.1	Javis . . . . .	68
7.2.2	Grading by annotation . . . . .	69



<b>A Student problems</b>	<b>71</b>
<b>B Annotation occurrences</b>	<b>75</b>
<b>Bibliography</b>	<b>77</b>



# List of Figures

1.1	Graduated students relative to admitted students the theoretical starting year. . . . .	2
3.1	Inspected student solutions. . . . .	18
3.2	Decrease of submitted solutions throughout the semester. . .	19
3.3	Problem taxonomy . . . . .	20
4.1	The most common problems among the students. . . . .	33
4.2	Students who had parameter passing problems. . . . .	34
4.3	Students who had problems with conditional expressions. .	36
4.4	Domain model (UML class diag.) of semester assignment 2.	37
4.5	Students with problems from the concept-based and knowledge-based categories. . . . .	38
5.1	Javis with a Java program loaded. . . . .	44
5.2	Simplified domain model (UML class diag.) of Javis . . . . .	46
5.3	<i>Scope stack</i> example . . . . .	48
5.4	AST of <code>int number = 5 + 3;</code> . . . . .	50
5.5	Example of step by step evaluation. . . . .	53
5.6	Nested scope example . . . . .	54
5.7	Example of parameter passing . . . . .	55



# List of Tables

6.1	File annotations . . . . .	60
6.2	Meta annotations . . . . .	61
6.3	Variable problem annotations . . . . .	62
6.4	Conditional expression problem annotations . . . . .	62
6.5	Method call problem annotations . . . . .	63
6.6	Scope problem annotations . . . . .	64
6.7	Control structure problem annotations . . . . .	65



# Chapter 1

## Introduction

In this chapter we explain the background and motivation for our work, present our research goals and give an outline of the thesis.

### 1.1 Background and Motivation

Learning to program is a difficult task that many students struggle with and fail to complete. Programming courses are generally regarded as difficult and many first year programming students perform much more poorly than hoped for [18] in these courses, resulting in high dropout rates [9, 24]. Our department offer two bachelor programmes, Computer Science<sup>1</sup> and Computer Technology<sup>2</sup>, and compared to the number of students admitted to the programmes, very few students graduate the scheduled graduation year (see Fig. 1.1).

In the capacity as a teaching assistant in the introductory programming course (INF100<sup>4</sup>) at the University of Bergen, the author has observed the struggle of the students first hand. By assisting the students at lab sessions and grading their submitted solutions to assignments, the author has observed a multitude of problems and difficulties. This motivated us to investigate this more in depth, and we especially wanted to see if some problems were as widespread as they seemed to be, and if particular problems persisted throughout the semester.

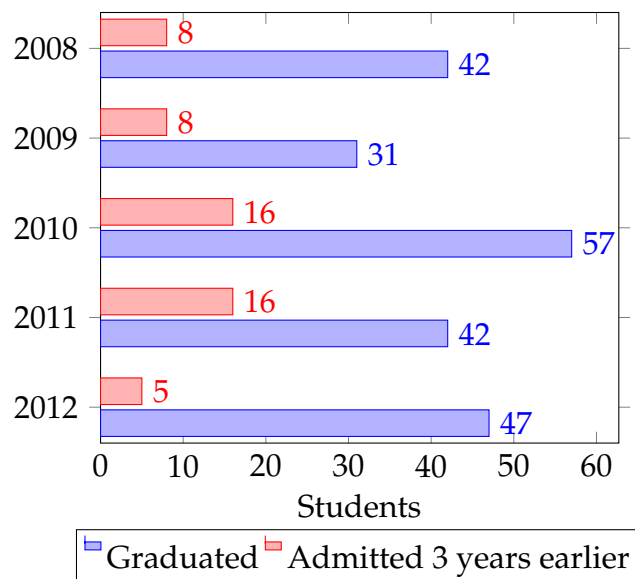
---

<sup>1</sup><http://www.uib.no/studyprogramme/BAMN-DVIT>

<sup>2</sup><http://www.uib.no/studyprogramme/BAMN-DTEK>

<sup>3</sup>The University of Bergen

<sup>4</sup><http://www.uib.no/course/INF100>



**Figure 1.1:** Number of students that graduated each year (2008-2012) relative to the number of students admitted to the study programmes the theoretical starting year (UoB<sup>3</sup>).

## 1.2 Research Questions

Our plan is to study the problems that students learning Java as their first programming language struggle with. We aim to identify and study mistakes, misconceptions and unsolved requirements in the solutions that the students submit to assignments given in the introductory programming course at the University of Bergen. Studying these problems, we then want to find out if:

- some problems are more common than others.
- particular problems, such as parameter passing and references is as common a problem as our experience suggest.
- problem occurrences can be related to misconceptions or missing knowledge.

Based on the experience and knowledge we gain in that study, we intend to develop a proposal of how to grade student assignments and collect data about problem occurrences, without increasing the workload or degrading the quality of the feedback to the students. This grading scheme can provide the teachers with valuable knowledge, as they then are able to address the most common misconceptions among the students.



Students often have incorrect conceptions of programming concepts [5, 16]. *Our hypothesis is that a significant part of these misconceptions can be observed using the right techniques.* Du Boulay et al. [10] suggests that how the programming language affect the computer should be supported with some kind of concrete tool which allows the interaction to be observed. We support this and believe that with an easy to use tool with proper visualisation of how programs actually work, step-by-step, the students themselves will be able to identify and understand these kinds of problems. We intend to develop such a tool.

### 1.3 Related Work

Lathinen et al. [14] analyses a survey of students and teachers opinions of which elements they consider difficult when learning programming. Parts of this survey is strongly related to what we look at in this thesis, though we have chosen other methods of gathering data and we have a more narrow focus. Robins et al. [24] reviews much research regarding learning and teaching of programming and present a discussion of the findings. Spohrer and Soloway [27] present an empirical study of the general belief of novice mistakes.

Johnson et al. [12] developed a bug classification scheme that relate bugs to program constructions and underlying misconceptions and Spohrer et al. [26] created a problem-dependent classification scheme based on Johnsons scheme. Our grading scheme (Chapter 6), and its taxonomy (Section 2.3), has similarities with both classification schemes, it relates problematic code segments both to the problem description (requirements) and the language constructs. We then discuss and connect problems with plausible underlying misconceptions.

### 1.4 Thesis Structure and Outline

To introduce the problem domain of this thesis we start with a discussion of the learning process of programming, where we look into the skills that the students must learn and the challenges that they will meet in the process. Based on that discussion and previous research regarding classification of student problems, we specify a student problem taxonomy that separates problems based on their underlying reasons. The two categories of problems that we look into are those that can be related to observable misconceptions, *concept-based*, and those that can be related to missing knowledge or

understanding, *knowledge-based*.

To answer the research questions of this thesis we inspected Java programs developed by students, and annotated all the problems that we found. We then study the data generated by the annotations to identify the most common problems of the students, and classify them in our taxonomy. With the classification of the most common problems we aim to verify that a significant part of the students' problems are *concept-based*, which we argue that the students can identify and understand themselves, with the proper assistance. We then look into particular problems and study the annotation data in detail.

To aid the students in identifying and understanding the *concept-based* problems, we have developed a visualisation tool which we present, and provide example executions of.

We then present a proposal of a grading by annotation scheme, that is based on the work in the other parts of this thesis.

### 1.4.1 Outline

A brief description of the contents of each chapter:

**Chapter 2: Learning programming** In this chapter we first discuss the skills that the students must master when learning programming, and the challenges they will encounter in the process. Then we specify a taxonomy that we have used to classify the most common problems of the students. We end the chapter with a description of the curriculum and organisation of the introductory programming course taught at our department.

**Chapter 3: Study design and Problem taxonomy** In this chapter we first give a description of the design of our study, and describe the data set used. Then we provide a thorough description of each of the problems we found were most common, and a list of plausible underlying reasons for them.

**Chapter 4: Result analysis** In this chapter we analyse the results of our study. First we describe the total amount of data we gathered, then we present and discuss our findings.

**Chapter 5: Tool assistance – Jarvis** In this chapter we present a tool that we are developing for students. We describe the requirements and intentions of the tool, and provide some examples of its use.

**Chapter 6: Grading by annotation** In this chapter we present a grading by annotation scheme that we have developed. We based it on the experience gained during the research for this thesis, and we describe how to use it, extend it and collect the data.

**Chapter 7: Conclusion** In this chapter we give a status summary of the work in this thesis, and we briefly explore the possibilities and future work of the *grading by annotation* scheme and *Javis*.



# Chapter 2

## Learning Programming

Learning programming is a complicated task that requires the student to acquire a lot of new and interconnected knowledge. This makes the teaching of programming a serious challenge for educators. With respect to the curriculum, different teaching methods, order of the topics and so on, they have many choices. Even the seemingly simple choice of which language to teach is a tough one:

- Should one teach programming using a dedicated learning language, or one in use in the industry?
- From which paradigm should one choose the language; functional, imperative or object-oriented?
- If object-oriented; should imperative (procedural) features or objects be taught first?

We gave this brief introduction into the teaching of introductory programming to inform the reader that the teaching methodologies of programming are disputed. We do not consider this further in this thesis, as we take another perspective on the challenges of teaching and learning programming, instead we refer the interested reader to the extensive research in this area. For example Robins et al. [24] for a review and discussion on learning and teaching programming, Dale [8] for a survey on CS1 content practices, Schulte & Bennedsen [25] for a study of what is taught in CS1 and Bailie et al. [4] for a panel discussion of the object first approach.

In this chapter we discuss what goals the student should reach during the learning process and the difficulties the students encounter along the way. Then we present and discuss the problem taxonomy that we use in this thesis to classify the typical problems students have. Following that we

give a thorough description of the CS1 equivalent course, INF100, offered at our department.

## 2.1 Programming Skills

The end-goal of students learning programming are very different, and depends primarily on what they are planning to use the skill for. Some are interested in applying the skill within academia, some as developers, others learn out of curiosity. No matter what end-goal though, it is fair to say that the student should be able to express solutions for typical computer-solvable problems in the language he or she is learning.

First off, this requires that the student must learn the language itself, that is, the syntax and semantics of the necessary language features (constructs). By necessary features, we argue that it is not required that the student learns *all* the features of the language initially, as there are many (depending on the language) operators, keywords and constructs that are not immediately needed. Typically bitwise operators, exception-handling constructs, generics, polymorphism and inheritance are such more complicated and not initially needed features. Though some languages may force you to teach them early, e.g. Java forces exception-handling quite early.

In addition to learning the semantics of the language constructs, the student should also learn typical ways of applying them. In literature this is often referred to as schemas, or sometimes plans, depending on the author [24]. We only use the term schema, and we use it to refer to a certain way of combining language constructs to solve specific problems, i.e. a schema may be a way of combining language constructs to calculate the sum of the numbers in a list (array) or read and validate user input etc. These trivial schemas will give the student both a better conception of the construct semantics, and knowledge of actual usage.

The next basic skill is the ability to properly interpret a problem description and generate a general notion of how to solve the problem. We do not however, recommend writing down an explicit solution in natural language and then translating that into the programming language. Spohrer and Soloway [27] argue that this may be a pitfall of possible misconceptions, as some natural language words are used as keywords for constructs in the programming language and do not necessarily have the expected semantics. Using this notion, or mental image, the student should then be able to express a procedural plan of how to apply a combination of schemas in a sequence that ultimately solves the problem. Depending on the size and complexity of the problem, it might be necessary to first divide it into smaller

problems.

Linn and Dalbey [15] propose that given a good learning environment the student should achieve the following “chain of cognitive accomplishments”:

1. Language features; understanding the semantics of the language structures.
2. Design skills; the ability to plan procedurally, apply schemas/plans, test and reformulate code.
3. Problem-solving skills; application of knowledge and strategies, in a way that is abstracted from the specific language being taught, i.e. able to apply it with other languages and settings.

It is possible to teach these skills separately without depending on the previous skill, e.g. teaching a schema for calculating the average of a list of numbers without using programming language examples. However we argue that the order given in this chain is important and should be followed by educators. This will provide the students with the means to experiment and observe that the schema works when applied using the respective constructs.

The ACT models by Anderson [2] suggest that abstract representations of knowledge cannot be learned directly, they can only be learned by practicing the operations on which they are based. The process of experimenting and observing consequences, learning-by-doing, is very important when learning programming and should be a primary focus. This will give the student a feel of accomplishment and improve the student’s conception of how it works.

Though we do recommend selecting subsets of constructs and follow the chain individually for them, e.g. selecting the if-statement, variables and user input calls, providing the student with schemas for these and skills to solve user input validation problems and actions based on the input. This way the student is able to achieve the full chain of knowledge for individual subsets quite early and gain experience with them.

## 2.2 The Challenges

In the process of learning programming the students will encounter many difficulties and challenges. Du Boulay [10] describes a list of five overlapping problem domains, that we describe in detail below:

1. *General orientation.* What programs are, and how they work.

2. *The notional machine*. How a language interacts with the computer.
3. *Notation*. Language syntax and semantics.
4. *Structures*. Possess and understand schemas.
5. *Pragmatics*. Planning, developing, testing, debugging etc.

**General orientation (1)** The understanding of what programs are, that they can be used to solve problems and what kinds of problems are solvable. Within the learning environment students are provided with problems to solve, these are implicitly solvable, and we argue that this is a challenge that may become more challenging later, outside the confines of the learning environment.

**The notional machine (2)** Du Boulay [10] describes this as:

An idealized, conceptual computer whose properties are implied by the constructs in the programming language employed.

That is, having a correct understanding of how the language constructs affect the computer, e.g. correct mental models of how an array is stored and accessed in memory, how references work etc. It is important to note that different languages have different models, though some are similar. This may be one of the most challenging parts of programming. The student may have a model that works in many cases, or even most, but have some cases where it breaks down and behaves unexpectedly. It is very difficult for the student to identify mistakes that are caused by this, since the entire conception of how the constructs work is tied to this model. It is improbable that the student even will suspect that the reason a section of the program does not work as intended is because the model that the student has observed to be correct in all previous cases really is incorrect.

**Notation (3)** The correct understanding of the syntax and semantics of the language. That is being able to express the language constructs syntactically correct and understanding the semantics of them. Any proper IDE<sup>1</sup> will provide the student with valuable assistance in the form of syntax highlighting, code formatting and active syntax error reporting. The error reporting removes some of the need for the student to interpret compile errors (which may seem very cryptic). The IDE reports errors by highlighting

---

<sup>1</sup>Integrated Development Environment



the location and providing an error message for any syntax errors. Syntax highlighting (color and emphasis) and formatting will make the code easier for the student to read and differentiate between language keywords, literals, variables etc., making the whole task of writing syntactically correct programs a lot easier.

Spohrer and Soloway [27] conclude that misconceptions about language constructs does not seem to be as widespread as is generally believed. Though the semantics of the constructs may prove a bit challenging at times, the student should be provided with good explanations from the course literature and lectures.

**Structures (4)** Knowledge of schemas that can be applied to solve small sub-goals of a problem, when and where to apply them and how to combine them. Understanding a schema properly, i.e. being able to apply it for problems not entirely the same as those given as examples, may prove difficult, and combining or nesting them; even more difficult. It requires the student to understand that the schema is a general solution, and understand the inner-parts of it, and being able to mold it to the problem at hand.

**Pragmatics (5)** Planning, developing, testing, debugging etc. are skills that are developed with experience and will be a challenge throughout the process of learning programming.

We use some of these problem domains in Section 2.3 where we specify a problem taxonomy for student mistakes. Problems in the *notional machine* domain and parts of the problems in the *structures* domain are related to misconceptions that are important for the *concept-based* category. While problems in the *notation* domain and other problems in the *structures* domain are related to missing knowledge and fall into the *knowledge-based* category.

## 2.3 Student Problem Taxonomy

The mistakes of students are many and diverse, and though they on the surface may seem to be the same problem, e.g. by having the same consequence and appear in an equivalent location, there may be very different underlying reasons. The same can be said vice versa, some mistakes may have the same root cause but appear in different locations, or have different consequences. This makes the classification of student mistakes quite complicated and

require a lot of interpretation.

In this section we present the taxonomy scheme that we use in this thesis. We base our work on that of Spohrer and Soloway [27] who built a taxonomy for student problems where they classified mistakes into two categories, *construct-based problems* and *plan composition problems*. These categories they describe as not mutually exclusive or exhaustive. To determine which category problems should be placed in, they identified plausible underlying causes for the problems and used them as determinants.

We use some of the causes that Spohrer and Soloway identified, refine some of them and provide additional ones that we have identified, as determinants for our taxonomy. However, we have chosen other categories, *concept-based problems* and *knowledge-based problems*.

Brooks [6] describes the program comprehension process as expectation driven by creation, confirmation and refinement of hypotheses. This is what we attempt to capture with the *concept-based problem* category. Problems that can be identified and rooted out by creating hypotheses and confirm or refine them.

### 2.3.1 Concept-based problems

Typical *concept-based problems* are those that are related to incorrect schema applications or misconceptions of the notional machine. Problems we classify as *concept-based* are those where the student should be able to understand the mistake by observing the behaviour of the program. With a combination of reading, manually tracing the code and observing the results of executing the program, while experimenting with the data set, the student should be able to locate and correct these mistakes. Instances of these problems are expressions, statements or constructs in a program that are either unnecessary and does not affect the result of the program, or in some way prevent the program from working as it should.

- I) *Incorrect notional machine*. An incorrect notional machine may have unexpected obscure consequences that may be challenging to notice and understand. Though difficult, they are identifiable by observing the behaviour of the program.
- II) *Natural language translation*. The student may devise a solution to a problem in natural language. Translating from natural language to the programming language is not straight forward, and may cause problems for the student. Observing the behaviour of the program,

the student should be able to identify that it behaves differently than expected.

- III) *Misconception of construct semantics.* The student may possess some conception of construct semantics that is incorrect, but seems to be correct based on previous experience. This may be an almost correct conception, that has some special case(s) it does not cover, or it may be based only on trivial cases and fail when more complicated parts of the semantics matter, e.g. evaluation order. These mistakes are observable, as the student has a correct intention, but failed to apply the construct due to some misconception.
- IV) *Specialisation problem.* The student may find it challenging to apply the general schema correctly to a concrete problem. When customising the schema to the particular situation at hand, the student may deviate slightly from the correct application and create an incorrect solution. These mistakes should be observable as the student has a correct understanding of the schema and knows what is intended.
- V) *Test data.* The data set the student is executing the program with may not be good enough to expose the problem, and if it was supplied with the assignment, the student has no reason to suspect that it is insufficient.

### 2.3.2 Knowledge-based problems

*Knowledge-based problems* are those situations where the student does not have the proper knowledge or understanding to solve a problem. This may be due to lack of schemas to apply, or an inadequate understanding of them (thus not able to apply them). Instances of these problems may be segments in a program that the student has either left completely blank or placed an empty skeleton of a schema. Other examples of instances may be a dynamic problem solved in a static way, with the consequence that they only work for the given example data set.

- I) *Interpretation of specification.* Correct interpretation of the assignment specification may be challenging. Explicit requirements may be neglected or perceived incorrectly and implicit requirements may be missed.
- II) *Schema knowledge.* The student may not possess the necessary schemas or adequate conceptions of the schemas to solve the problem.

- III) *Existing environment*. Students may find it difficult to express their intention using the relevant existing parts of the program environment, e.g. identifying the correct variables and methods to use.

## 2.4 INF100 Course

INF100, our CS1-equivalent course, gives a thorough introduction to programming and is a mandatory course in both bachelor degrees offered at our department. The course, or an equivalent course, is a prerequisite for all other programming courses taught at our department. Though it is primarily aimed at computer science students, it is also offered and even recommended to students of other natural science degrees, and especially to Mathematics and Physics. This means that there is a lot of diversity in the students background, skill and motivation, and that the progress of the course must be adjusted accordingly. The primary motivation of the course is to teach students how to use a programming language to solve problems with a computer and doing so in a general sense. By that we mean that even though the course teaches one language, namely Java, it aims at doing it in such a way that the students learn the “programming way of thinking” and hence will be equipped with the knowledge to learn and use other languages as well.

### Curriculum

We now provide a chronological list of the curriculum of INF100, which is reflected in both the lectures and the assignments given in the course.

1. *Imperative language features*. The first part, about one third, of the curriculum focuses on the basic concepts and constructs of imperative languages, e.g. variables, data types, expressions, control flow, arrays, input/output and assignment. The students will in addition be in contact with, and use, objects in this part, which is due to Java’s strong object-oriented nature. That means that a quick introduction to objects and data abstraction is required (and given) in this part, as well.
2. *Objects*. About one third into the curriculum the focus will shift towards object-oriented-programming, where object communication, inheritance and polymorphism will be introduced. This is primarily a theoretical introduction and little to no focus will be given inheritance and polymorphism in the course assignments.

3. *Algorithms*. About half way into the curriculum algorithms, e.g. searching and sorting, will be introduced, followed by abstract data types and dynamic data types, e.g. linked lists etc.
4. *Recursion*. Then recursion and recursive algorithms are presented, which also is given little to no attention in the assignments.
5. *Exception handling*. Towards the end exception handling is introduced both in the lectures and in assignments, followed by file- and stream handling.
6. *GUI*. Graphical user interfaces may be given a short introduction, if there is interest and available time for it.

## **Organisation**

There is a heavy emphasis on learning by doing in the course, which is reflected by the many mandatory assignments and available lab sessions. We strongly encourage the students to attend at least one of the five lab sessions offered each week. If there is room, they may attend multiple sessions. In these sessions it is envisaged that the students will work on their current course assignment, and it is encouraged that they co-operate and help each other. At each session a teaching assistant will assist the students with their problems, and preferably in an educational way. The assistants are all talented and experienced students who have come further in their studies, many at the end of their bachelor degree and some are working on their master thesis. There are however no special requirements to pedagogical skills or education, though the Psychological faculty offer them a course in pedagogy.

In total there are ten assignments, of which seven are exercises and three are larger mandatory semester assignments. The exercises are only graded with a pass or fail, though the teaching assistant grading it will provide comments for the work. Of these, the student must pass at least five, but is required to submit something for all of them – even if it is an empty submission. The larger semester assignments are all graded with an individual score, where the total score of all three counts 30% towards the final grade in the course.



## Chapter 3

# Study Design and Problem Taxonomy

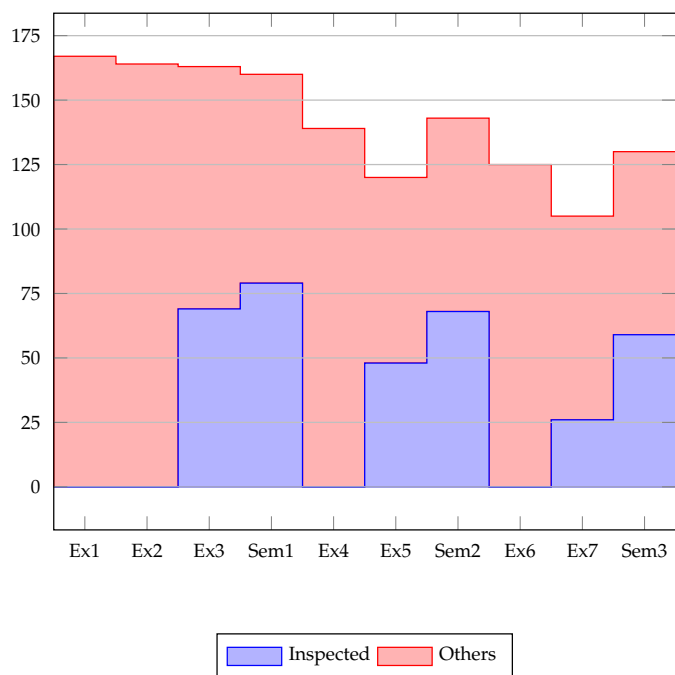
The goal of this study is to identify and study the specific problems that students may encounter while learning programming, and the frequency of them. To achieve this we performed an empirical study by inspecting the source code written by the students for the assignments given in the INF100 course. We were interested in all kinds of problems and solutions, both syntactic and semantic.

In this chapter we first describe how we collected the data, and the size of the data set. Then we describe and classify some of the most frequent occurring problems using the taxonomy described in Section 2.3.

### 3.1 Study Description

The data we collected had to be represented in such a way that we would be able to analyse it in many different ways, without the need to inspect the programs multiple times. We found that by annotating any interesting expressions, statements or segments with an identifier, we could scan the files for these annotations afterwards and insert the information into a database. A database provided us with a powerful query language that we could use to retrieve the data and build interesting queries with. Each entry in the database contains a student identifier (anonymous), assignment, file, exact line where the problem occurs and the problem identifier. Storing the data in this way also enabled us to verify the results later, extract code snippets and make the data available for later research.

Before we began inspecting the student submissions we created an initial list of typical student problems, based on our previous educational work,



**Figure 3.1:** Inspected student solutions.

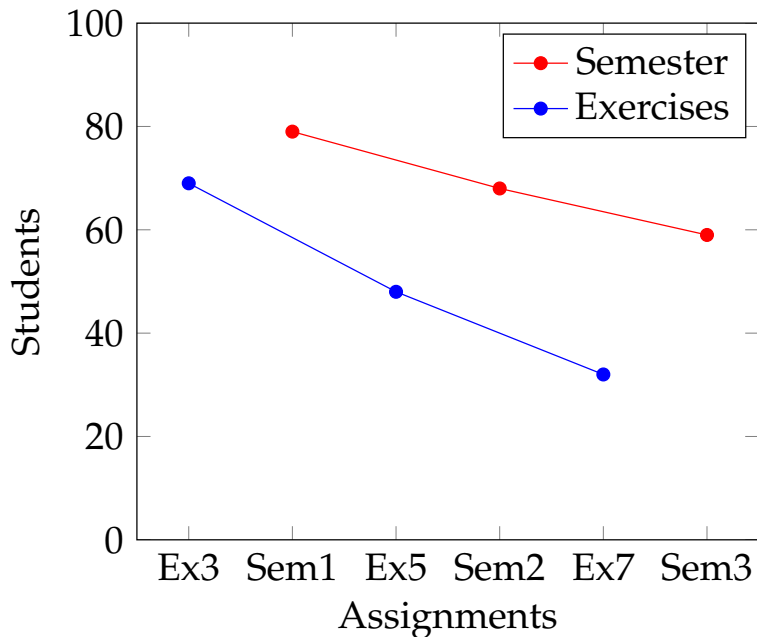
and gave them each a unique identifier. However, during the process of annotating the source code we discovered other kinds of problems, that we had not initially foreseen, and also had to change some of the initial ones as they were either too general or too specific. See Appendix A for a complete list of the annotations used.

## 3.2 Data Set

We have a large number of submissions available from just one semester of the course INF100. In total 182 students enrolled for the course and there were 3 semester assignments and 7 exercises during the course. This sums up to a total of 1820 submitted solutions, if all the students had submitted a solution for all the assignments. However since a portion of the students dropped out during the course, and they were only required to pass 5 out of the 7 exercises, there was slightly fewer submissions, approximately 1400.

This was still a large number of submissions, and we did not have the time and resources to inspect all of these. We had to limit the set of submissions to inspect (see Fig. 3.1) and we began by selecting a subset of the students, based on their performance on the semester assignments, of





**Figure 3.2:** Decrease of submitted solutions throughout the semester.

which we would inspect the submitted assignments. Any student scoring strictly less than 25 out of 30 ( $\sim 85\%$ ) points on any of the three semester assignments would be added to the set. The reason why we chose this limit was that most of the submissions scoring higher than 25 points had very few mistakes, and they were mostly superficial.

This gave us a set of 79 students, which still was a too large amount of data for us to inspect. We then proceeded to limit the data set by selecting six assignments to inspect. The first two exercises were trivial introductions to programming, and had little value to us as they only involve simple arithmetics in a *Hello World* way. The three semester assignments were sure choices, and to close the time gap between them we chose the three exercises that were given before each semester assignment.

In Fig. 3.2 we present the number of students, from the subset we selected, who submitted solutions for the selected assignments. The decrease of students who submitted solutions for the semester assignments are all students who dropped out of the course, i.e. 20 students dropped out between semester assignment 1 and 3. For the exercises however, the decrease is mostly because the students are only required to submit solutions to five of them. Exercise 7 was especially affected by this.

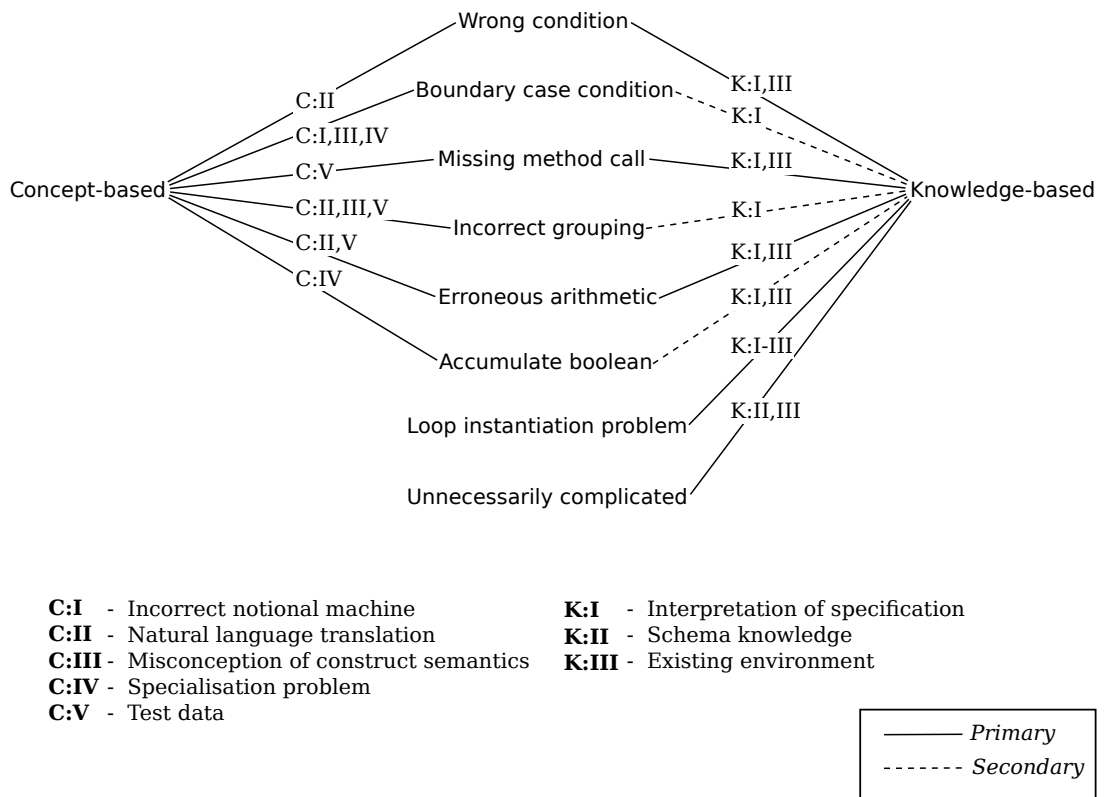


Figure 3.3: Problem taxonomy

### 3.3 Problem Taxonomy – Most common problems

In this section we give a thorough description of the most frequent occurring problems and then we classify (see Fig. 3.3) them using the taxonomy described in Section 2.3. Together these problems account for more than half of all problem occurrences (see Section 4.7) and by classifying these we may be able to suggest actions that can be taken with respect to the teaching. For each problem we provide a general description, concrete examples and, if necessary, actual code samples. To classify the problems we have identified a list of plausible underlying causes for each problem and an argument is given for why those causes are plausible. Appendix A is a valuable resource throughout this section.

#### 3.3.1 $\text{WRNG}_{\text{COND}}$ – Wrong condition

Any conditional expression that in some way is incorrect may be counted as a *wrong condition* problem. However the annotation is mutually exclusive

with those annotations that consider special cases of incorrect conditional expressions, e.g.  $\overset{\text{BDRY}}{\neq}_{\text{COND}}$ . Except for  $\overset{\text{BDRY}}{\neq}_{\text{COND}}$ , we did not identify any other special cases of conditional expression problems that were common enough to deserve their own annotation. Though we do recommend adding annotations for any kind of conditional expression, e.g. tautology conditions, that seems to be common.

Occurrences of this problem are primarily incorrect conditional expressions in if-sentences, though there are a few occurrences in loop-constructs. Typical occurrences of this problem are conditions that are wrong because they use an incorrect operator, connective or method call. Concrete examples of this problem are solutions that check for reference equality instead of object equality, or overcomplicated conditions that render the solution incorrect, etc.

We suggest these plausible underlying reasons:

1. *Interpretation of specification.* The students may find it difficult to extract the correct conditions from the specification. It requires them to understand the given specification in its entirety, and they may miss the implicit conditions.
2. *Natural language translation.* Not all logical expressions expressed in natural language can be directly mapped to the programming language. Conditions may change meaning and parts may be lost in translation.
3. *Existing environment.* The student may find it difficult to combine the relevant existing state of the program into a correct conditional expression. This often leads to overcomplicated or conflicting expressions.

The underlying reasons we identified for this problem are determinants from both the concept-based problem and the knowledge-based problem categories. Though the majority of the determinants are for the knowledge-based problems, we argue that it is uncertain what category is primary and secondary and choose *both* categories.

### 3.3.2 $\overset{\text{BDRY}}{\neq}_{\text{COND}}$ – Boundary case condition

Any conditional expression that evaluates incorrectly for some boundary case, either by going out of bounds or being too restrictive, e.g. incorrectly included or excluded boundary elements in a range. A concrete example of this is a condition that uses the inclusive less-than operator instead of the strictly less-than. This might result in an attempt at accessing an index

outside of an array, or a guard that accepts too little or too much of a number range. As a consequence the program might suffer of index-out-of-bounds errors for arrays. Listing 3.1 is a typical example of this kind of problem.

Listing 3.1: Boundary case sample

```
1 int[] numbers = new int[10];
2 for(int i = 0; i <= numbers.length; i++){
3     numbers[i] = i;
4 }
```

We suggest these plausible underlying reasons:

1. *Specialisation problem.* Identifying the correct boundary points may be challenging for the student when instantiating a schema.
2. *Incorrect notional machine.* The student may possess an incorrect conception of how the conditional expression relates to the environment, e.g. a model where the initial array-index is one-based instead of the actual zero-based.
3. *Misconception of construct semantics.* The student may have an incorrect understanding of how the language construct works, e.g. at what point the condition of a loop is validated or when the update statement of a for-loop is executed (or even what kinds of expressions that should be used as update statements). Du Boulay [10] notes that the automatic increment of the counter variable may be problematic.
4. *Interpretation of specification.* Students may have interpreted the program specification incorrectly, e.g. not perceived correctly if it should be an inclusive or exclusive condition.

These reasons are mostly determinants in the concept-based problem category, though (4) belongs to the knowledge-based problem category, and we argue that this problem should be classified primarily as a *concept-based problem*, and secondarily as a *knowledge-based problem*.

### 3.3.3 $\hookrightarrow_{\text{CREATE}}^{\text{LOOP}}$ – Loop instantiation problem

Any situation where the student has had trouble instantiating a loop-construct to solve a given task, is counted as a loop instantiation problem. Occurrences of this problem are situations where:

- The student has either not been able to instantiate a loop, or did not realise that one was needed, often leaving that part unsolved or given a hard coded solution.

- The student has only partly instantiated a loop, often the most trivial loop-construct schema.
- The student has instantiated a loop that is far away from a working solution, often with conditions and bodies that does not resemble a solution in any way.

Though situations where the student has solved the problem in a static manual way are not counted, these are counted as  $\frac{5}{8} \frac{MAN}{LOOP}$ . Listing 3.2 is an example of a case where the student has either not been able to instantiate a loop or did not properly understand the semantics of the existing environment and failed to see why line 7 would not compile.

Listing 3.2: Missing loop X

```

1 //Field variable representing shares in percentage
2 double[] shares = {30.0,25.0,10.0};
3 /**
4  * @return the remaining share in percentage.
5  */
6 public double getShareOfOther(){
7     double otherShare = 100 - shares;
8     return otherShare;
9 }
```

We suggest these plausible underlying reasons:

1. *Schema knowledge.* The student may not possess the necessary schemas to instantiate a loop construct to solve a particular problem, or may not understand how the schemas work and is not able to use them for any other cases than the default.
2. *Existing environment.* To combine the existing knowledge of the program and use the state of the environment when instantiating a loop construct may be very difficult. To realise which methods/functions and/or variables that should be utilised, requires a good overview of the program and a good conception of the semantics of them.
3. *Interpretation of specification.* In some cases the requirement that specifies the need for a loop-construct may be less obvious and the student may miss it.

All the underlying reasons we identified for this problem are determinants from the knowledge-based category, and we argue that it should be classified as a *knowledge-based problem*.

### 3.3.4 <sup>CMPLX</sup><sub>CTRLS</sub> – Unnecessarily complicated

Instances of this problem are those situations where the student devise an overcomplicated solution that may or may not work. These code segments may be very difficult to understand, even for the student at the time of writing, and if it should prove to not work as intended it is very difficult to identify exactly what is incorrect. Many instances of this problem could in some situations, with a more thorough inspection and communication with the student, have been annotated with other more specific annotations. Typically these are complicated structures that may be any combination of the following (or other less frequent examples):

- Deeply nested blocks.
- Repetition of large segments of code.
- Conditional structures with individual branches for each possible case the student can imagine, even though they may not be relevant or should have the same result/consequence.

Listing 3.3 is an example of an unnecessarily repeated code segment that could easily be removed with some arithmetic or by performing the conditional difference calculation first. This implementation does not follow the specification completely either, it misses the case where time1 is equal to time2.

Listing 3.3: Repeated code 

```
1  if(time1 > time2){
2      int difference = time1 - time2;
3      /*
4      ...
5      Code segment that prints out the difference
6      converted to hours, minutes and seconds.
7      ...
8      */
9  }
10 else if(time1 < time2){
11     int difference = time2 - time1;
12     /*
13     ...
14     Exact same code segment.
15     ...
16     */
17 }
```

We have identified the following plausible underlying reasons:

1. *Schema knowledge*. The student may not possess or understand the schemas necessary to instantiate sensible conditional structures, prevent repetition etc.
2. *Existing environment*. The student needs a good understanding of the environment, and its state, to be able to prevent the creation of unnecessary complicated structures. To prevent repetition the student needs to identify what separates the different cases, and how to exploit this knowledge to combine the segments.

Both the determinants we have identified and listed above are from the knowledge-based category, and we argue that it should be classified as a *knowledge-based problem*. However, in some cases there may be another underlying reason for this problem, namely if the student fears to change something that to some degree may fulfill the requirements.

### 3.3.5 $\zeta_{\text{CALL}}^{\text{NO}}$ – Missing method call(s)

Any situation where the student has omitted a necessary, or useful, method call is counted as a *missing method call* problem. In some situations the student has still managed to create a working solution by re-inventing the wheel, i.e. implementing the functionality of the method specifically for that situation, and in others the student may have left out that part of the solution entirely. When solving assignments given in a course, the student will in some cases be provided with methods (or classes) and/or information about any library methods that could be useful.

We have identified the following plausible underlying reasons:

1. *Existing environment*. The student needs a good overview of the existing program structure and the available library methods to be able to avoid this problem. Using this knowledge the student must be able to realise the possibility of calling a method in a given situation, and be able to identify which method.
2. *Interpretation of specification*. The student may not have perceived the specification correctly and in turn does not realise that there should be a certain method call in that situation.
3. *Test data*. The student may not be able to realise that there is a problem if the data set the student is executing the program with does not expose that the method(s) was not called.

Two of the determinants identified for this problem, (1) and (2), are for the knowledge-based category and the third is for the concept-based category. We argue that it is more likely that the student does not realise there is a missing method call due to insufficient test data than incorrect conception of the specification, and classify this as both a *concept-based problem* and a *knowledge-based problem*.

### 3.3.6 – Incorrect grouping

Any situation where the student has incorrectly connected, or failed to connect, constructs and/or statements is counted as *incorrect grouping* problem. Typical examples of this are:

- Incorrectly connected, or failed to connect, multiple if-constructs (using the else-connective).
- Omitted to place those statements that should be the “else-case” in an else-block, i.e. placed in such a way that they do not depend on the if-construct.
- Placed the result action of a search schema in the body of the loop, resulting in that the action that should be performed on the result of the search, is performed each iteration instead.
- Placed a statement that also should depend on the condition of an if-structure, outside of that block.

Listing 3.4: else connective

```
1 if(player1Winner){
2     System.out.println(‘‘something funny’’);
3 }
4 if(player2Winner){
5     System.out.println(‘‘an angry message’’);
6 }
7 else {
8     System.out.println(‘‘something sad’’);
9 }
```

We have identified the following plausible underlying reasons:

1. *Misconception of construct semantics.* The student may not properly understand how the if-construct connects using the else-statement, and consequently fails to connect them. This may be an understanding where sequential if-statements automatically connect (as if there was an else-statement connecting them) and that any else-statement at



the end of the sequence is the branch that is entered if none of the previous if-statement branches are entered.

2. *Natural language translation.* A plan of conditional actions may be expressed correctly in the natural language of the student, but in such a way that it does not directly map to the programming language. Listing 3.4 implements these requirements “If player1 wins print something funny, if player2 wins print something sad, otherwise print an angry message”. The student has implemented the requirements by directly mapping the natural language solution into the programming language and the angry message will only be printed if player2 does not win.
3. *Interpretation of specification.* It may be challenging for the student to perceive the specification as it is intended, especially regarding implicit requirements. In many cases the requirement that the calculated result of a program should not be printed if input validation fails, is often implicit and many students does not realise that.
4. *Test data.* The test data the student is using may not expose that there is a problem with the current structure, e.g. the data set might be so small that there is only one iteration of the search loop and the result action is only evaluated once.

The majority of the underlying reasons identified for this problem are determinants from the concept-based problem category, and we argue that this problem should be classified primarily as a *concept-based problem* and secondarily as a *knowledge-based problem*.

### 3.3.7 ⚡<sup>BAD</sup><sub>ARIT</sub> – Erroneous arithmetic

Any arithmetic expression that is incorrect with respect to the given assignment is counted as erroneous arithmetic. Common occurrences of this problem is when the student needs to calculate the difference between two numbers, and neglects that the result might be negative. The student may realise the problem but fail to find a correct solution, often statically multiplying by  $-1$  or finding the absolute values of the individual terms instead of the entire expression. Other occurrences of this problem may be that the student fails to understand exactly what the numbers represent, what denominator they have and/or what they count.

We have identified the following plausible underlying reasons:

1. *Test data.* The test data the student is using may not expose that there is a problem with the arithmetics, e.g. no case where calculating the difference will give a negative result.
2. *Existing environment.* The student may not possess the necessary understanding of the existing environment and fails to see the correct semantics of a number, e.g. incorrect denominator.
3. *Interpretation of specification.* The student may not have understood the specification correctly, e.g. choosing the wrong representation of a value.
4. *Natural language translation.* The experience the student has with arithmetics is likely purely static and the dynamic behaviour of arithmetical expressions in a programming language may be challenging and may make it difficult for the student to translate the expressions.

The underlying reasons identified for this problem are determinants from both the concept-based and knowledge-based problem categories, and we argue that it is uncertain if one of them is more dominant than the other, and choose to classify this problem as both a *concept-based problem* and a *knowledge-based problem*.

### 3.3.8 <sup>BOOL</sup><sub>ACCUM</sub> – Accumulate boolean

Instances of this problem are situations where the student has neglected, or failed to, accumulate a result boolean when performing an operation on a list of elements. The operation that is performed on each element returns a boolean value that represents failure or success. This boolean value should be accumulated by the loop-construct iterating over the list of elements in a way that detects any failure, see Listing 3.5.

Listing 3.5: Boolean accumulate 

```

1 boolean success = true;
2 for(Recipe r: getRecipes()){
3     if(!r.printToFile())
4         success = false;
5     //or the reduced way
6     success = success && r.printToFile();
7 }
8 System.out.println("Success? " + success);

```

We have identified the following plausible underlying reasons:

1. *Existing environment*. The student may not realise that the operation performed on the elements returns a boolean value representing its success, and fails to realise that there should be an accumulation.
2. *Interpretation of specification*. The student may not have perceived that the resulting boolean expression should consider the operations.
3. *Specialisation problem*. The student may not have been able to express how to accumulate the boolean correctly.

The majority of the identified underlying reasons are from the knowledge-based category, (1) and (2), though we argue that (3) is more likely because the student is syntactically forced to have a boolean return expression in the method implementation. This should provide the student with a reason to investigate how that value should be calculated, which should limit the number of students having (1) and (2) as underlying reasons. We argue that it is primarily a *concept-based problem* and secondarily a *knowledge-based problem*.



# Chapter 4

## Result Analysis

In our study we inspected and annotated the source code of 349 solutions submitted by the 79 selected students, for the six selected assignments. A total of 2395 annotations across 920 files. Of these 1185 are meta annotations that describes if the files:

- do not compile ( $\neg \overset{\text{COMP}}{\underset{\text{ERR}}{\downarrow}}$ ).
- contain methods that are not implemented ( $\overset{\text{UNSOLV}}{\underset{\text{PROB}}{\downarrow}}$ ).
- are not implemented at all ( $\neg \overset{\text{NOT}}{\underset{\text{IMPL}}{\downarrow}}$ ).
- sufficiently solve the problem, or displays sufficient understanding of how ( $\overset{\text{PASS}}{\downarrow}$ ).
- do not sufficiently solve the problem ( $\neg \overset{\text{FAIL}}{\downarrow}$ ).
- are changed in a way that breaks the premise of the assignment ( $\neg \overset{\text{CHNG}}{\underset{\text{EXER}}{\downarrow}}$ ).

We did not have strict requirements for the pass annotation ( $\overset{\text{PASS}}{\downarrow}$ ), so any file where the student was able to display how the problem should be solved to a satisfying degree, was annotated. Even some files that did not compile (42) was annotated with pass. The pass ( $\overset{\text{PASS}}{\downarrow}$ ) and fail ( $\neg \overset{\text{FAIL}}{\downarrow}$ ) annotations are mutually exclusive. There were only 94 files annotated with fail, and the rest, 808, was annotated with pass.

There were 1210 problem annotations divided among 52 different annotation kinds. Some of these annotations had very few occurrences and are not necessarily referred to, and if they are, it is in relation to some other problem that they are relevant for.

### 4.1 Lazy students?

While we studied the gathered data we noticed that as many as 49 of the 79 students had at least one file annotated with  $\overset{\text{UNSOLV}}{\underset{\text{PROB}}{\downarrow}}$ , missing implementation

of a method. In total 83 files was annotated with  $\frac{\text{UNSO}}{\text{PROB}}$  and in 17 files there were multiple occurrences (i.e. multiple missing methods). In addition 14 students had files that were not implemented at all,  $\frac{\text{NOT}}{\text{IMPL}}$ , in total 20 files. We can only share our suspicions of why there were such a significant part of the submitted solutions that was not implemented. In our opinion there are two reasons that seems to be the most plausible.

1. The student may have speculated in how much of the assignment that were necessary to implement to achieve a passing grade.
2. The student did not possess the knowledge to be able to provide a solution for the missing parts.

If lazy students (1) was the reason why some of the submitted solutions were missing parts, we may have reasons to suspect that this may have been an underlying reason for other problems identified in the student submitted solutions as well. This especially applies to those situations where the solution was missing some part, e.g.  $\frac{\text{LOOP}}{\text{CREATE}}$  (loop instantiation problem). Though it may be possible that the student would have chosen to implement the skipped parts if the student had enough experience and knowledge such that the implementation would require less work and time.

We give this argument to caution the reader that the following analysis may be affected by this, and to make the reader aware that we did have this in mind when we performed the analysis.

## 4.2 Most common problems

In Fig. 4.1 we present a chart of the most common problems among the students, i.e. those problems that the most students were registered having. Conditions, overcomplicated solutions, loop-constructs and method calls are those problems that, by the data we collected, were the most common problems. In the chart there are also some annotations regarding unnecessary expressions, statements and placement:

- $\frac{\text{XTRA}}{\text{VAR}}$ , unnecessary variable.
- $\frac{\text{PRE}}{\text{DECL}}$ , unnecessarily pre-declared variable.
- $\frac{\text{BOOL}}{\text{EQL}}$ , checking boolean expression for equality with boolean literal.

These three are only indicators that shows that the student may have some misconception of the semantics or the notional machine, and we have only gathered them under the idiom *problem annotation* for convenience. The

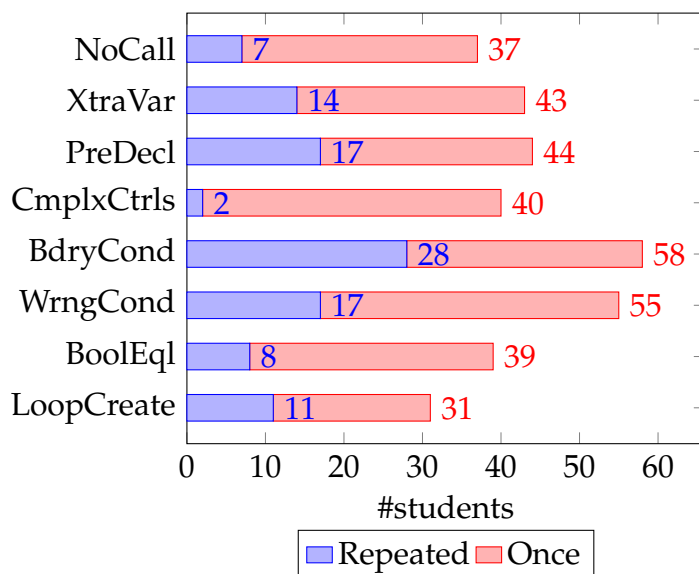
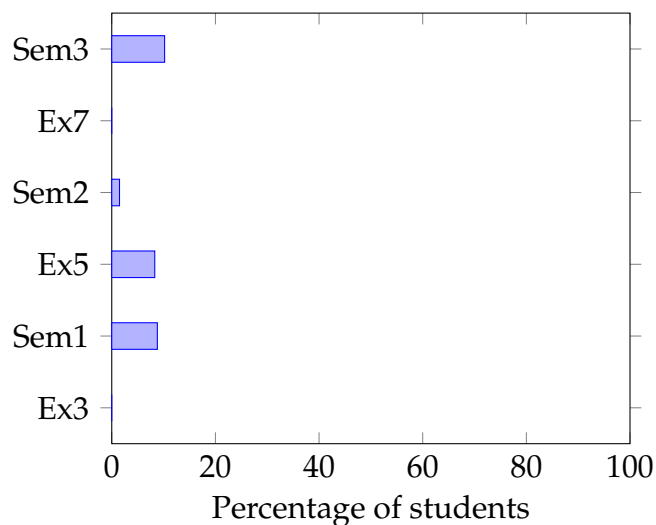


Figure 4.1: The most common problems among the students.

students are quickly rid of the  $\frac{30}{7}$   $\frac{BOOL}{EQL}$  and  $\frac{10}{7}$   $\frac{PRE}{DECL}$  misconceptions, most likely due to good feedback from the teaching assistants, as they are primarily only a “problem” in semester assignment 1 (and exercise 3 as well for  $\frac{8}{7}$   $\frac{PRE}{DECL}$ ). The occurrences of  $\frac{10}{7}$   $\frac{XTRA}{VAR}$  are more thinly spread across the assignments, though most frequent in exercise 5 and semester assignment 2. We argue that the most plausible reason for this “problem” is that it is some remnant of some plan or previous attempt to implement a solution, and that the student most likely has forgotten the existence of the variable.

Conditional expressions is a concept that our data shows was a big challenge for many students. The most common kind of conditional expression problem was incorrect boundary case conditions,  $\frac{28}{7}$   $\frac{BDRY}{COND}$ , which as many as 58 students were registered having problems with. In addition 55 students were registered having trouble with conditions in general,  $\frac{17}{7}$   $\frac{WRNG}{COND}$ , that is those that does not fall into a special category (like boundary case). Most of the students (48) who had troubles with boundary case conditions also had troubles with conditions in general. Respectively 28 and 17 students were registered with  $\frac{28}{7}$   $\frac{BDRY}{COND}$  and  $\frac{17}{7}$   $\frac{WRNG}{COND}$  across multiple assignments.

We registered 40 students that created overcomplicated solutions,  $\frac{2}{7}$   $\frac{CMLX}{CTRLS}$ , though only two students created such solutions in multiple assignments. In many cases it is plausible that the reason that these kinds of solutions are submitted is that the student fears to change something that



**Figure 4.2:** Students who had parameter passing problems ( $\zeta_{\text{ASSGN}}^{\text{WRNG}}$ ,  $\zeta_{\text{VARS}}^{\text{GLBL}}$ ,  $\zeta_{\text{SCOPE}}^{\text{ARG}}$ ,  $\zeta_{\text{PARAM}}^{\text{FRML}}$ ).

seems to work, even though it might be clear that the code segment is very complicated and/or repetitive. We also suggest that it is plausible that this is a problem that students have with newly learned concepts, and that with experience they will learn to apply the concepts properly.

### 4.3 Parameter passing and references

Our experience in teaching suggested that the concepts of parameter passing and references would be a significant and common problem in the submitted student solutions. However this was not the case, and in fact very few students were registered as having difficulties with parameter passing, and none were registered having troubles with references. See Fig. 4.2 for the frequencies of parameter passing problems.

The teaching assistants reported that parameter passing was a hot topic in their lab sessions. If we consider this in relation to the low frequency of this problem in the submitted solutions, we may come the conclusion that the teaching assistants were successfully able to teach the students this concept. Though there is also the possibility of assistance from other students.

There is one interesting case though ( $\zeta_{\text{PASS}}^{\text{PARAM}}$ ), 22 students seem to have a small misconception regarding how parameter passing works. In some cases when they write method call statements, they declare an



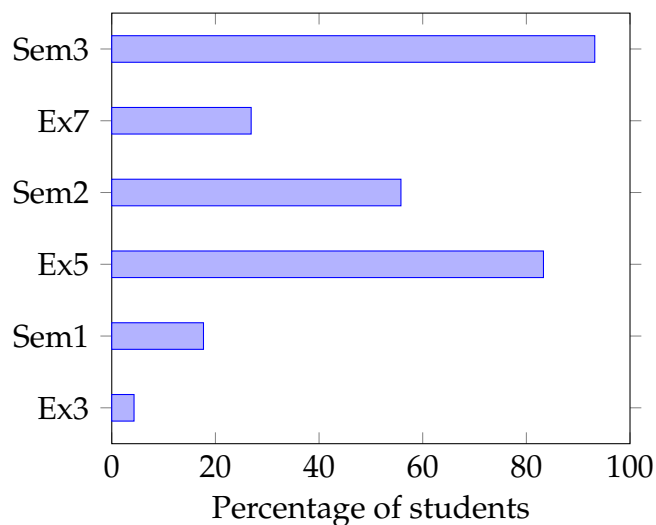
unnecessary variable on the line above the method call with the same name as the formal parameter of the method. In the last two assignments the complexity of the domain model increased significantly, which we suggest is the reason why this problem only occurs in those assignments (.....) and why the frequency is high.

We suggest that a reason for the lack of reference problems and the low frequency of parameter passing problems might be related to the design of the given assignments. Situations where references could have been a problem in the assignments were all related to immutable objects like String, Integer etc.

## 4.4 Conditions


Problems related to conditional expressions, both in if-sentences and loop-constructs, was very common among the students, see Fig. 4.3. Semester assignment 3 was the most difficult assignment given during the course, and not only did it stress most of the previously learned topics, it also required the students to implement a total order relation-method (compareTo from the Comparable interface). Almost all, 93.2%, of the selected students who submitted a solution for this assignment had some trouble with conditional expressions. If we also take into account the problem of accumulating a boolean value ( $\frac{6}{7} \text{ ACCUM}^{\text{BOOL}}$ ), there was only one student who did not have problems. Exercise 3 and semester assignment 1 both had an introductory focus on if-sentences and loop-constructs, though the students were required to interpret, and understand, specifications for these structures, and instantiate them in the implementation. In these assignments few students had difficulties and most were able to express correct conditional expressions. We argue that this may show that the students are able to identify situations where these structures are required and at least able to interpret the concrete specifications of the conditions. But when the specifications and conditions grow more complex in the later assignments the students experience difficulties.

A large part of the solutions submitted for exercise 5 and semester assignment 2 also had mistakes related to conditions, respectively 83.3% and 55.8%. The listed goals of these assignments are respectively *Objects and methods* and *Classes and objects*, but there are few identified problems related to these – which may suggest that the actual focus of the assignment was incorrect.



**Figure 4.3:** Students who had problems with conditional expressions ( $\zeta_{COND}^{BDRY}$ ,  $\zeta_{COND}^{XTRA}$ ,  $\zeta_{IF}^{UGLY}$ ,  $\zeta_{COND}^{WRNG}$ ,  $\zeta_{CORRCT}^{ALMST}$ ).

## 4.5 Loop constructs

Implementing programs that requires loop constructs is a challenge for most novices, and bugs are more common in relation with these constructs than with other common tasks like input, output, syntax/block structure and overall planning [26]. This is reflected in our results as well, during the course 82.2% of the selected students had troubles related to loop constructs, either with the condition or the instantiation of the construct. The assignments where the most students struggled with loop constructs was exercise 5 and semester assignment 2 and 3, .

**Semester assignment 2** In this assignment 74% of the selected students that submitted a solution struggled with loop constructs. The focus of this assignment was *Classes and objects* but there was a frequent need for loop-constructs, and the nesting between the provided Java classes was quite deep for a novice. This may have caused the students to have difficulties when creating the mental model of how the classes and objects communicate. The given classes were `Client`, `Recipe`, `RecipePart` and `Ingredient`, see Fig. 4.4. A `Recipe` object has a list of `RecipePart` objects, which in turn has a specific `Ingredient` object associated with it and the `Ingredient` object has an array of nutritions. In addition the main-method in the `Client` class has a list of `Recipe` objects. This provides many locations where loop-constructs are needed to traverse the data structure, and even some where the need is not

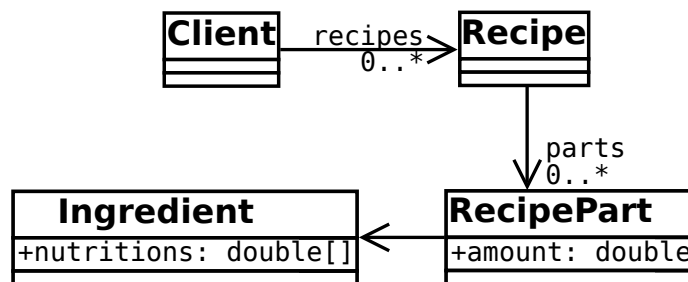


Figure 4.4: Domain model (UML class diag.) of semester assignment 2.

specified directly, which may explain why the problem is more common in this assignment than others.

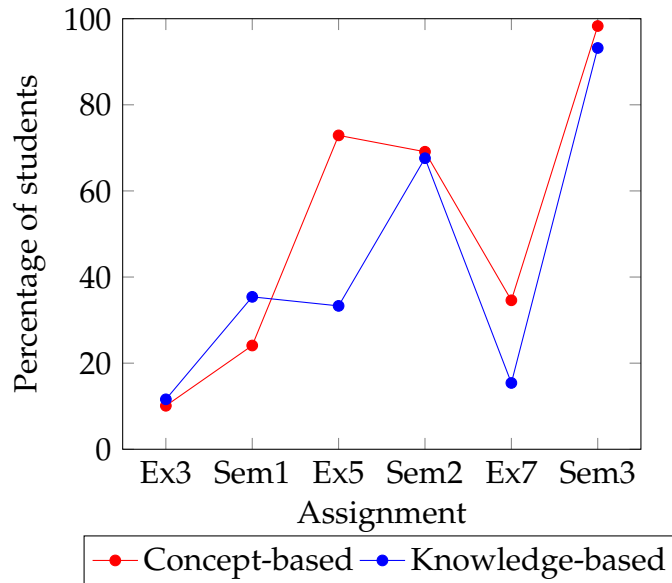
## 4.6 Method calls

During the course 60.1% of the selected students had misconceptions or difficulties with method calls. These problems were most common in semester assignment 3, where 59.3% of the selected students who submitted a solution had troubles. Most of the problems that the students had were situations where a necessary method call was missing ( $\cancel{\text{CALL}}^{\text{NO}}$ ), which 46.8% of the selected students struggled with. In many of these cases the students had re-implemented the responsibility of the method by writing a copy of the method body (presumably unknowing) at the location where the method call should have been, and thus met the requirements of the assignment. In other cases the program did not perform the task of the method, and did not meet the requirements.

There was also other problems related to method calls, though quite rare. A few students struggled with correctly specifying method calls, where both syntax and choosing which expressions to pass as arguments ( $\cancel{\text{CALL}}^{\text{STATIC}}$  and  $\cancel{\text{CALL}}^{\text{BAD}}$ ) were problematic. Other cases were more subtle and did not affect the solution in any way, e.g. repeated calls to observer methods ( $\cancel{\text{CALL}}^{\text{XTRA}}$ ,  $\cancel{\text{VAL-S}}^{\text{UNUSD}}$  and  $\cancel{\text{VAL-O}}^{\text{UNUSD}}$ ).

## 4.7 Concept-based problems

If we disregard the annotations that signify unnecessary expressions, statements and placement, then the most common problems, both in occurrences and the number of students that had them, were those that we described in



**Figure 4.5:** Students with problems from the concept-based and knowledge-based categories.

our taxonomy, Section 3.3. As mentioned previously there was in total 1210 occurrences of *problem annotations*, of these 314 was of the unnecessary kind (described in Section 4.2) – yielding 896 problem occurrences. The most common problems accounted for 560 of these occurrences, which is more than half of the problem occurrences. Below we have listed the problem annotations by the classification specified in Section 3.3 (disregarding the secondary category).

- *Concept-based problems.*  $\frac{207}{7}$  BDRY COND,  $\frac{83}{7}$  WRNG COND,  $\frac{54}{7}$  NO CALL,  $\frac{204}{7}$  WRNG GRPNG,  $\frac{86}{7}$  BOOL ACCUM,  $\frac{57}{7}$  BAD ARIT.
- *Knowledge-based problems.*  $\frac{83}{7}$  WRNG COND,  $\frac{40}{7}$  LOOP CREATE,  $\frac{202}{7}$  CMLPX CTRLS,  $\frac{54}{7}$  NO CALL,  $\frac{57}{7}$  BAD ARIT.

Three of these problems ( $\frac{83}{7}$  WRNG COND,  $\frac{54}{7}$  NO CALL,  $\frac{57}{7}$  BAD ARIT) are in both categories, and thus 61 students are associated with both categories – 66 students in total are associated with both. Individually there are 70 students associated with each category. In Fig. 4.5 we present, for each assignment, the percentage of students who had problems from the two categories. Notice that for most assignments, all except for exercise 3 and semester assignment 1, there were more students who had *concept-based problems*, and there was slightly more occurrences of problems from this category as well, 435 against 358. Based on the results that we have presented it is not possible to conclude that one of the categories is more important than the other, though that is not

our intention either. What we can conclude is that the *concept-based problem* category is an important one, and that improving the environment that the student has available to discover and learn about these mistakes likely will yield better results.

## 4.8 Conclusion

The research goals that we set in Section 1.2 were:

1. Are some problems more common than others?
2. Are parameter passing and reference problems as common as our experience suggested?
3. How do problems relate to misconceptions and missing knowledge?

We have found that some problems are more common than others, and that the two most common problems are related to conditional expressions,  $\frac{\text{BDRY}}{\text{COND}}$  and  $\frac{\text{WRNG}}{\text{COND}}$ . These are also most prone to repetition.

Parameter passing and references are not as common as we expected, at least not for the assignments that we inspected in this study. Though we do suspect that this may be because the assignments are not particularly challenging regarding these subjects, and that it would be a more common problem if the assignments had given this more attention. However more attention on these subjects may translate to less attention on other subjects.

To relate problems to misconceptions and missing knowledge we used the taxonomy we described in Section 2.3. The category *concept-based problems* cover problems that are related to misconceptions, and the category *knowledge-based problems* cover those related to missing knowledge. We found that there were many occurrences of problems from both categories, that they are both important and that we cannot argue that one is more important than the other. This confirms our hypothesis that a significant part of the problems are related to misconceptions.



# Chapter 5

## Tool Assistance – Javis

In Section 4.7 we concluded that a significant part of the problems identified in the students submitted solutions were *concept-based problems*. An important property of the problems that we classify as concept-based, is that the students themselves can observe and identify them using the correct techniques. Du Boulay et al. [10] suggest that a concrete tool that students can observe their mental model (*notional machine*) with, is useful. Ma et al. [17] found evidence suggesting that a teaching model based on visualisation can help students develop correct mental models of programming concepts.

Experienced programmers use a combination of unit tests, debuggers and manual tracing of code to identify and fix bugs of this sort [24]. For many students learning their first programming language these techniques are a significant challenge, and at a time when learning the language syntax and semantics is enough of a challenge on its own. “Novices are frequently poor at tracing/tracking code”, Perkins et. al. [23].

In this chapter we specify and describe a tool, Javis<sup>1</sup>, that we are developing, which is designed specifically to assist students learning Java as their first programming language. With Javis, we aim to provide the user with an easy way to debug and visualise the execution and evaluation of Java programs. We emphasise that it is aimed at students learning the elementary language features, and that it will not be feature complete with the Java Virtual Machine, nor with a full-featured debugger.

### 5.1 Requirements

In order for Javis to be accessible to all potential users it is not to be tied to any specific platform nor require an installation, as most computers available

---

<sup>1</sup><http://www.javis-tool.org>

at educational institutions have limited user access rights. Typically many introductory Java programs communicate with the user through a command line interface (CLI), this may be an overwhelming interface for students. To encourage use of Jarvis, from the very beginning, it must be intuitive to use, have a small learning curve and offer a less frightening interface than a CLI.

One of the most important features of a debugger is breakpoints and step by step execution of statements, Jarvis provides this in a simpler form. We believe that step by step evaluation of statements, and their sub-expressions, without the use of breakpoints will provide a user interface that is easier to understand.

The lack of the breakpoint feature may in some situations lead to tedious interaction with the tool. To remedy this Jarvis should provide the user with the ability to adjust the precision of the step by step evaluation, i.e. which kinds of expressions to visualise or not to visualise the evaluation of. In addition the user should be able to execute individual methods, and be prompted in an intuitive way to supply the arguments for the called method. Allen et al. [1] suggest this ability to call individual methods, without writing a block of code for it in the main-method, may prove to be a valuable feature for beginner programmers.

Some of the basic language features of Java will not be adequately visualised by step by step evaluation and highlighting, e.g. parameter passing, scopes, method calls, flow of loop constructs, variables and references. If these are not properly visualised the users will be left relying on their own conception of how they work. The visualisation of correct semantics is one of the primary intentions of Jarvis, and these language features must be visualised in a satisfactory way.

List of requirements:

1. Non-functional requirements:

- (a) *Must* have a simple and intuitive user interface.
- (b) *Must* require no special knowledge to use.
- (c) *Must* be platform (o/s, software etc.) independent.
- (d) *Must* require no installation.
- (e) Only *need* to provide an entry-level support of the Java language features.

2. Functional requirements:

- (a) *Must* provide a step by step evaluation of expressions and sub-expressions.



- i. The user *should* be able to set the precision of the evaluation visualisation.
- (b) *Must* provide a view for text output through `System.out`, and input through `System.in`.
- (c) The user *must* be able to provide argument expressions for method calls (e.g. the `String` array for the main-method).
- (d) *Should* provide the possibility of calling individual methods, without executing the main-method.
- (e) There *should* be an intuitive visualisation of special language features.
  - i. Parameter passing
  - ii. Scopes
  - iii. Method calls
  - iv. Flow of loop-constructs
  - v. Variables, distinction between primitive and reference values.

## 5.2 Implementation

We are developing Jarvis as a web-application written in Dart [11], and the source code is available at GitHub<sup>2</sup>. A Dart web application requires no installation for the user and is platform independent, which conforms to the requirements *1c* and *1d*. Dart is a language that is currently still in development, by Google, and targets modern desktop and mobile browsers. Dart web applications can run natively in a Dart virtual machine or be compiled to JavaScript. The Dart to JavaScript compiler is aware of the differences between the JavaScript engines in use today and takes that into account when compiling. A significant benefit of using a client-side language like Dart is that it requires little resources to provide the service. There are other languages that offer similar properties as Dart, such as TypeScript<sup>3</sup> and CoffeeScript<sup>4</sup>, but the benefits between them are heavily debated and there is little literature on the subject – the choice is merely left to preference. The user interface depends only on HTML5 (Fig. 5.1).

---

<sup>2</sup><http://github.com/mapster/JavaEvaluator>

<sup>3</sup><http://www.typescriptlang.org>

<sup>4</sup><http://coffeescript.org>

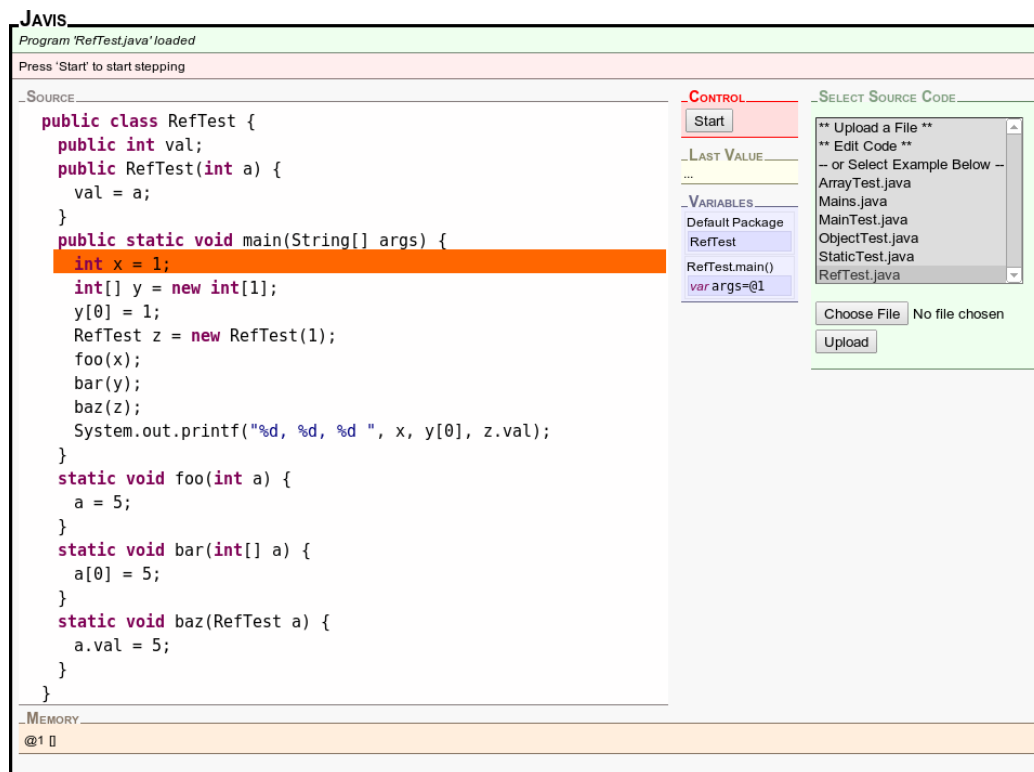


Figure 5.1: Javis with a Java program loaded.

## 5.2.1 Java evaluation

We have developed our own Java evaluator in Dart for Javis. Developing our own evaluator provides us with full access to the entire environment and stack of the executing Java program. The requirements that we have set for Javis specifies that it should visualise scopes, parameter passing, method calls, and provide a step by step evaluation and the possibility of calling individual methods (requirements 2e, 2a and 2d), which all are difficult to achieve without this access.

A significant challenge with implementing our own Java evaluator is achieving complete compliance with the Oracle JVM. However Javis only needs to support entry-level Java language features (requirement 1e). Additionally we can argue that it is not necessarily important to comply completely with the Java Specification, as long as the language construct semantics are the same. For example, the representation of primitive number types will in most situations not be an important requirement, as the evaluator is only intended to execute novice programs and the difference of representation will not matter, except if the student or an exercise is testing

the limits of for example the float type. We have however designed the evaluator in such a way that it is possible to achieve this requirement later if needed.

Security issues, and the no installation requirement (1d), also influenced our decision to develop our own evaluator. If Jarvis were to depend on a JVM to debug student submitted Java programs, it would require either using a client-side or a server-side JVM. Server-side execution of unverified and untrusted source code, that potentially is malicious, could result in severe security threats for the service provider. Executing untrusted data on an interpreter is first on OWASP's<sup>5</sup> 2013 Top 10 list of critical web application security flaws. Client-side execution means providing a Java Applet in the web-browser, but that requires installation of the Java plug-in.

### 5.2.2 Abstract syntax tree

Javis relies on the *Java Compiler Tree API* [20] to parse and build the Abstract Syntax Tree (AST) used by the evaluator. We have developed a Java Web Service that through the *visitor pattern* provided by the *Compiler Tree API* transforms Java Source Code to an AST represented as JSON [7]. The source code of this service is available at GitHub<sup>6</sup>. This API provides us with an AST that is guaranteed to comply with the Java Language Specification [21], and any compile time errors and warnings will be available. Our evaluator has no type safety mechanisms, and relies entirely on this service.

## 5.3 Tool Design

The simplified domain model (class diagram) in Fig. 5.2 illustrates the most central classes of Jarvis, and their dependencies. To provide a general overview of Jarvis we have left out many important classes that represents the sub-elements of the environment, execution and the abstract syntax tree. The *Program* class represents the top-level of the AST of the submitted Java program, and has a list of *CompilationUnit* objects, which represents Java files. Each kind of node in the AST has a corresponding class in our AST representation. The objects in the AST representation are not changed during an execution, and can be used for multiple executions without rebuilding. The *Runner* class is the connection between the user interface and the execution of the Java program. It is initially responsible for

---

<sup>5</sup>[https://www.owasp.org/index.php/Top\\_10\\_2013-A1-Injection](https://www.owasp.org/index.php/Top_10_2013-A1-Injection)

<sup>6</sup><http://github.com/mapster/StaticAnalysis>

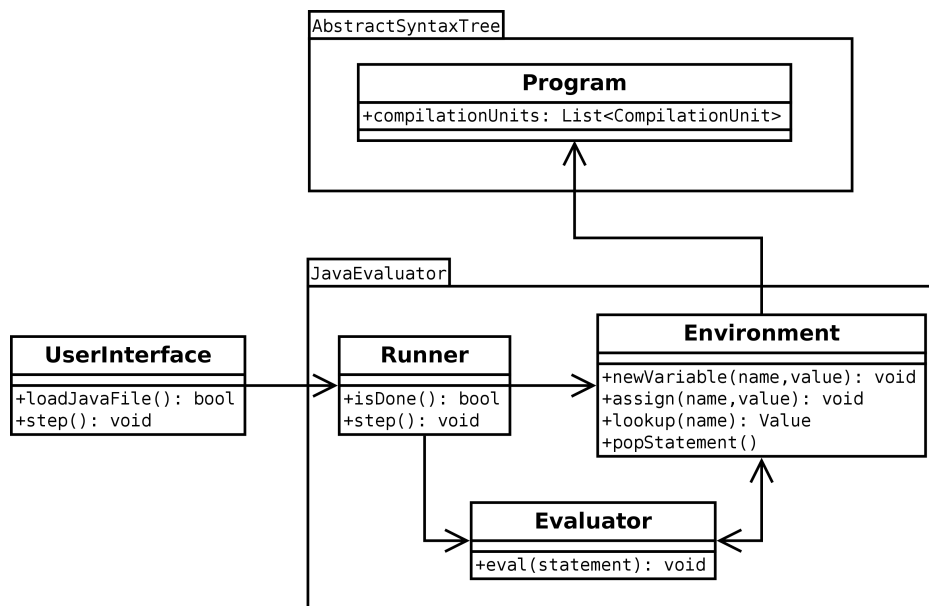


Figure 5.2: Simplified domain model (UML class diag.) of Jarvis

setting up the environment and loading the submitted classes, and during the execution it passes statements fetched from the environment to the evaluator. Method calls triggers loading of statements into the statement queue in the environment, which for all Java programs initially is a call to the main-method. The *Environment* class needs an association with the evaluator to be able to instantiate new objects of classes, this is because it needs to evaluate both the instance variables with initialisers and the constructor statements. The *Evaluator* class in turn has an association with the environment so that it can look up the *identifiers* and *member selects* that denotes method calls, variables etc.

### 5.3.1 Design concepts

In this section we describe important concepts in Java, and how we have represented them in Jarvis.

#### Namespace

Both Java packages and Java classes are *namespaces*, both represented internally by their own classes, that adheres to the following rules. Package names are given in a hierarchical manner such that a package may contain other packages, as well as classes. Classes may however only contain

definitions of other classes. The levels of the hierarchy are separated with the period character, e.g. the *ArrayList* class belongs to the `java.util` package and has the fully qualified name `java.util.ArrayList`. Each member in a namespace must have a distinct name within that namespace, e.g. there cannot exist both a class and a package named `String` in the same package. Compilation units (Java source files) that do not declare a package belongs to the *default package*. Members of the default package cannot be referenced outside that package since the package has no name.

### Identifier

An *identifier* is a node in the AST that represents a lookup in the current *namespace* and *scope*. The *identifier* have different meanings in different contexts, and may be a lookup for a variable, method, class etc. A *new object*-node may use an *identifier* to denote the class it is to create an instance of, this class must be available in the current *namespace*. Similarly a *method call*-node may use an *Identifier* to denote the method to call, this method must also be in the current *namespace*. And last but not least an *identifier* is in the most common case a lookup for a variable.

### Member Select

A *member select* is a node in the AST that represents a lookup into another namespace or scope. The node consists of two parts, the owner (or parent) and the member. The owner can be defined by either a *member select* or an *identifier*, but the member must be an *identifier*. A fully qualified class name used in a *new object*-statement is an example of a *member select*, that in many cases are trees of member select statements. The *member select* have different meanings in different contexts, and may be a lookup to instance variables or methods on an object, static variables or methods for a class, a class in a package hierarchy etc.

### Scope

The *scope* term denotes the context where the *identifier* of an entity is available for lookup. For example the scope of a variable declared in a method body is the part of the method-body that lexically follows the declaration. This variable will also be available in any sub-scope of the method-body, that lexically follows the declaration, e.g. the then-block of an if-statement.

A compilation unit may contain *import* declarations that denote, by a fully qualified name, a class that is made available in the *scope* of the

unit. It is also possible to declare an *import* that makes all the members of a package available in the *scope*, except for packages, e.g. `import java.util.*;`. In addition a third kind of import declaration exist, *static import*. These declarations make static members of a class available in the *scope*. It is however important to note that any member made available through import declarations are only available inside the scope of the compilation unit.

In the description of *Scope stack* we describe the internal representation of scopes.

### Scope stack

The *scope stack* is our representation of how nested scopes make out the complete scope for the next statement to be evaluated. In Fig. 5.3 the different gradients represent the different scopes, and the darker the deeper it is nested. For each scope in the stack all the brighter scopes are available. This should be familiar for most programmers. There are however, different kinds of scopes that follow different rules. For example, it is allowed to declare variables in method scopes that overshadow class level variables, but it is not allowed to declare variables in inner block scopes of method scopes (e.g. if blocks) that overshadows those declared in the method scope.

```
public class Test {
    static int a = 30;

    static void function(int b){
        int c = 5;
        if(true){
            int d = 3;
        }
    }
}
```

Figure 5.3: *Scope stack* example

### Call stack

The *call stack* is our representation of the stack of active methods. Every action in a Java program is somehow initialised by a method, with the main-method at the root. Method-call statements place a new method-body onto the stack. Each method placed onto the stack has its own *scope stack* associated with it. For example a static method has its parent class as root

of the *scope stack*, which makes any static variables and methods in that class available, second it has a scope for the method-body which initially contains all the formal parameter variables for the method, initialised with the arguments passed by the method call.

### 5.3.2 Evaluation

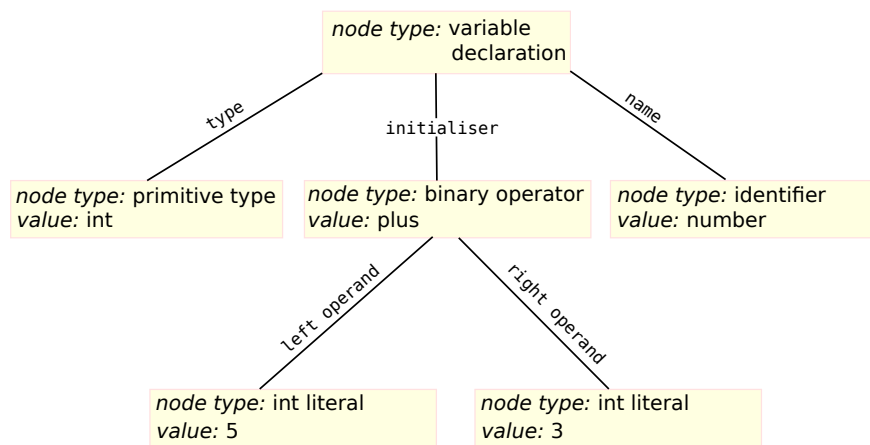
In this section we describe in detail how the evaluation is performed by Jarvis, and how it is represented internally. The class diagram of Jarvis in Fig. 5.2 may be helpful when reading this.

Evaluation of a step is triggered by a click event on the step button in the user interface, which the `step()` method of the `UserInterface` class is connected to. The method performs the tasks necessary to update the view of the user interface, and pass the control to the `step()` method in the `Runner` instance. The `Runner` instance then calls the `popStatement()` method in the `Environment` instance, and then passes the returned value to a call to the `eval()` method in the `Evaluator` instance.

Listing 5.1: EvalTree pseudo-code

```
1  class EvalTree {
2      List subExprs;
3      List ehaledExprs;
4      Function effectMethod;
5
6      dynamic execute(){
7          while(not subExprs.empty and
8              subExprs.first is Literal)
9              ehaledExprs.add(subExprs.removeFirst())
10
11         if(not subExprs.empty){
12             subExprs.first = subExprs.first.execute();
13             if(subExprs.first is not EvalTree)
14                 ehaledExprs.add(subExprs.removeFirst());
15
16             return this;
17         }
18         return effectMethod(ehaledExprs);
19     }
20 }
```

The `eval()` method accepts any statement, expression or literal node from the AST and pattern matches it on the node type. Literal nodes



**Figure 5.4:** AST of `int number = 5 + 3;`

are directly converted to the internal value representation and returned. For statement and expression nodes the Evaluator creates an internal intermediary representation of the node's expression tree using the `EvalTree` class (Listing 5.1). `EvalTree` has three field variables, `subExprs`, `evalExprs` and `effectMethod`. Where `subExprs` is a list of non evaluated sub-expressions, and `evalExprs` is a list of evaluated sub-expressions. The `effectMethod` is a Dart function closure that performs the task/effect of the statement or expression that an `EvalTree` object represents, e.g. if a variable declaration is represented then the method declares that variable in the environment. The `execute()` method performs the actual evaluation, with the first task of evaluating the elements in the `subExprs` list and moving them to the `evalExprs` list. Any literal nodes in the `subExprs` list is moved directly to the `evalExprs` list, without regarding it as a step. When the `subExprs` list is empty, it calls the `effectMethod` closure with the `evalExprs` list as the argument. An `EvalTree` object may require multiple steps, and will return itself until the evaluation is completed.

A concrete example where a Java program is loaded into Javis where the next statement in the statement queue of the environment is `int number = 5 + 3;` becomes as illustrated in Fig. 5.4. Only the variable declaration node and its initialiser node will be represented as `EvalTree` objects.

1. Step button click event.
  - (a) The statement is removed from the queue and passed to the `eval()` method.
  - (b) An `EvalTree` object is created to represent the variable declaration, with a Dart function closure that declares a variable in the



- environment with the name given by the child node *name*, and the child node *initialiser* as the only element in the *subExprs* list.
- (c) `execute()` is called on the `EvalTree` object, which has the *initialiser* node in the *subExprs* list.
  - (d) The `eval()` method is called with the *initialiser* node as the argument.
  - (e) An `EvalTree` object is created to represent the *binary operator* node, with a Dart function closure that adds the operands and returns the sum, and the *operand* child nodes in the *subExprs* list.
  - (f) `execute()` is called on the object, which has literal nodes in the *subExprs* list, these are immediately moved to the *evaluatedExprs* list.
  - (g) The `effectMethod` closure is called with the *evaluatedExprs* list as argument, and its return value (8) is returned.
  - (h) Back in the `execute()` method of the variable declaration `EvalTree` object, the return value of the *initialiser* object is moved from the *subExprs* list to the *evaluatedExprs* list.
  - (i) Control returns to the user interface.
2. Second step button click event.
- (a) Evaluation of the variable declaration object continues, and `execute()` is called.
  - (b) The *subExprs* list is now empty, and the `effectMethod` closure is called with the *evaluatedExprs* list as argument.
  - (c) The variable *number* is declared in the environment with the value 8.
  - (d) Control returns to the user interface.

## 5.4 Visualisation Examples

In this section we show some illustrations that Jarvis provides, or will provide, for the learning students.

### 5.4.1 Step by step evaluation

In Fig. 5.5 we illustrate the step by step evaluation that Jarvis provide. The illustration is of a simple Java program that only has one relevant scope

and no method calls, except for the initial main-method call. For more complex programs Jarvis provides a view that presents the entire *scope stack* of the method currently on the top of the *call stack*. The *scope stacks* for any other methods on the *call stack* are available for the user to expand as well. In the figure we have only included the evaluation of the first occurring assignment `x = 5;`, and left out the two subsequent ones in order to avoid repetition (step 4 and 10). By highlighting, Jarvis clearly visualise the evaluation of each sub-expression in the expression tree of the current statement. The result of the evaluation (of each expression) is presented to the right of the current statement in a distinct style to clearly visualise that it is a result of an evaluation, and to prevent the user from confusing it with the actual statement. If the expression is a variable then the corresponding identifier is highlighted in the *scope view* to the right. In the next step the result of the evaluation replaces the expression, again in a distinct style. This replacement is performed for any kind of expression, except for literals. When Jarvis has successfully completed the evaluation of a statement and moves on to the next, the altered statement is replaced with the original.

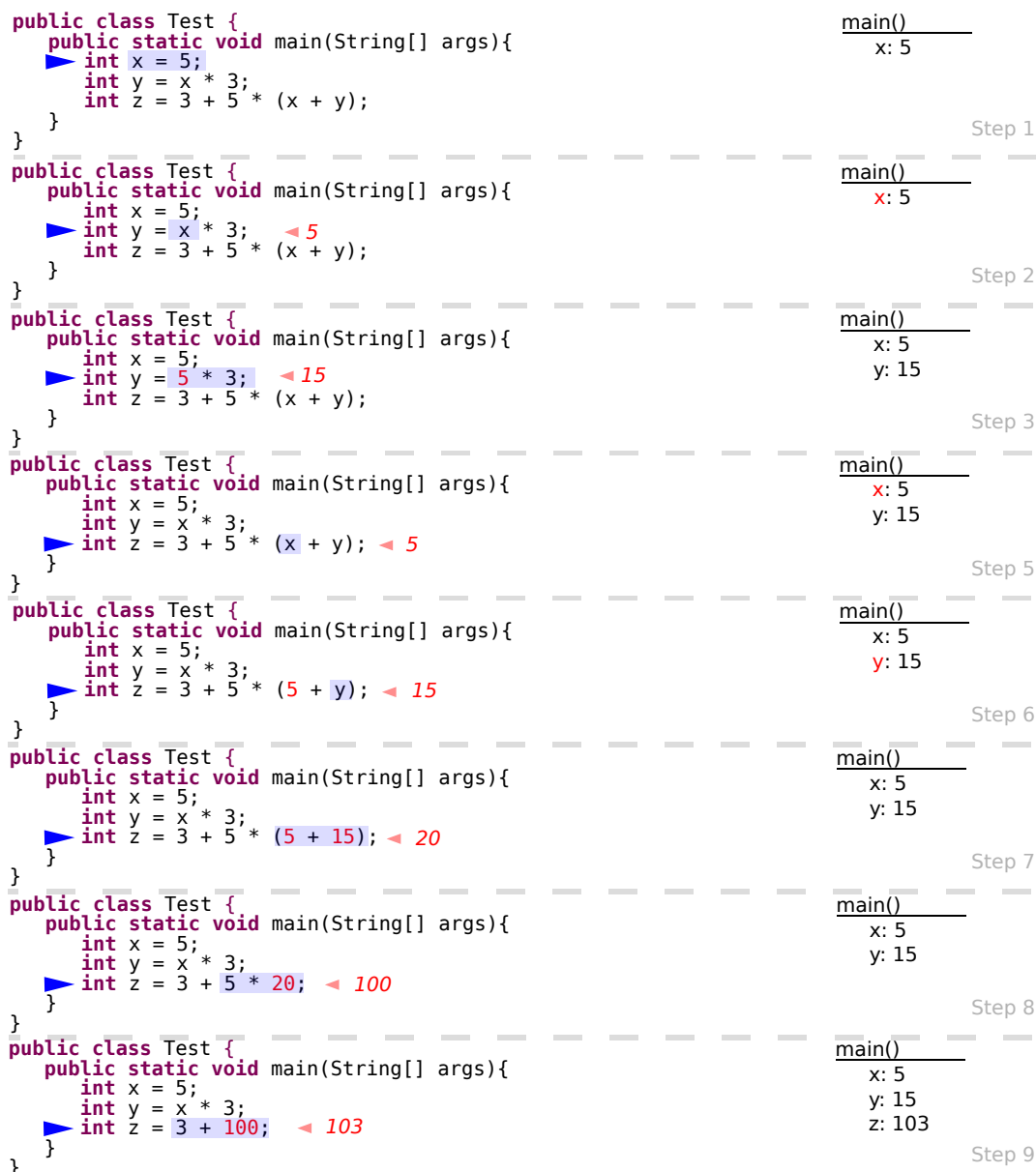


Figure 5.5: Example of step by step evaluation.

## 5.4.2 Nested scopes

Fig. 5.6 illustrate how Jarvis visualises the scopes of nested blocks. In *step 1* the variable `outer` is declared in the scope of the `main()` method, to help illustrate how the nested scopes are visualised. In *step 2* the condition of the `if`-statement evaluates to true, the then-block is entered and a new

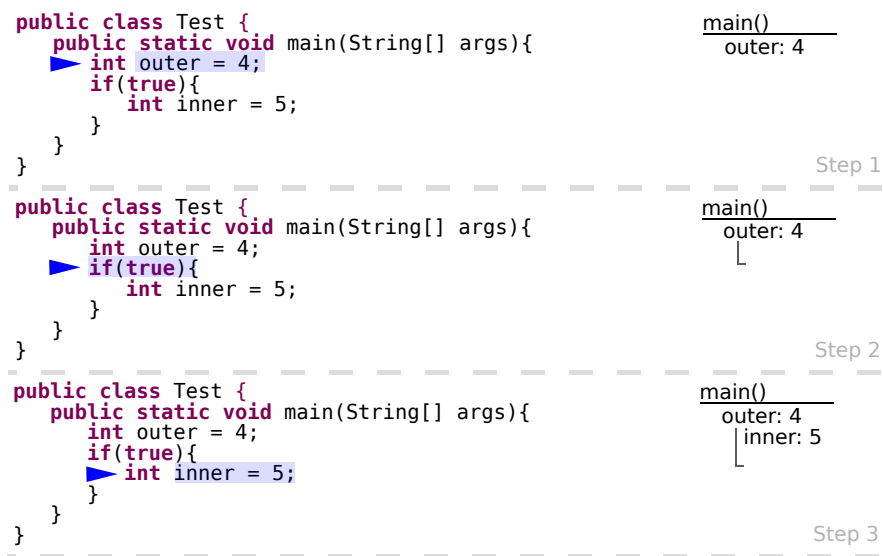


Figure 5.6: Nested scope example

block scope is put on the *scope stack* (visualised by the line below the outer identifier). The variable `inner` is declared in *step 3* and is displayed in the nested scope in the *scope view*.

### 5.4.3 Parameter passing

In Fig. 5.7 we illustrate how we have planned that Jarvis will visualise parameter passing for method calls. First, *step 1*, the tool visualise evaluation of all, if any, arguments in the method call statement. When all the arguments are evaluated the method call is evaluated and the corresponding method is highlighted. In *step 3* the visualisation of the actual parameter passing begins. In turn, stepwise, each formal parameter is highlighted for evaluation, an arrow is drawn between the argument and the formal parameter, and a corresponding identifier is listed in the *scope view*. The two following steps illustrate a statement that requires look up of the formal parameter in the environment.

## 5.5 Related Work

There exists several systems and tools that have similar goals and/or provide similar functionality to Jarvis. Among these systems BlueJ [13] and DrJava [1] is perhaps the most relevant ones.

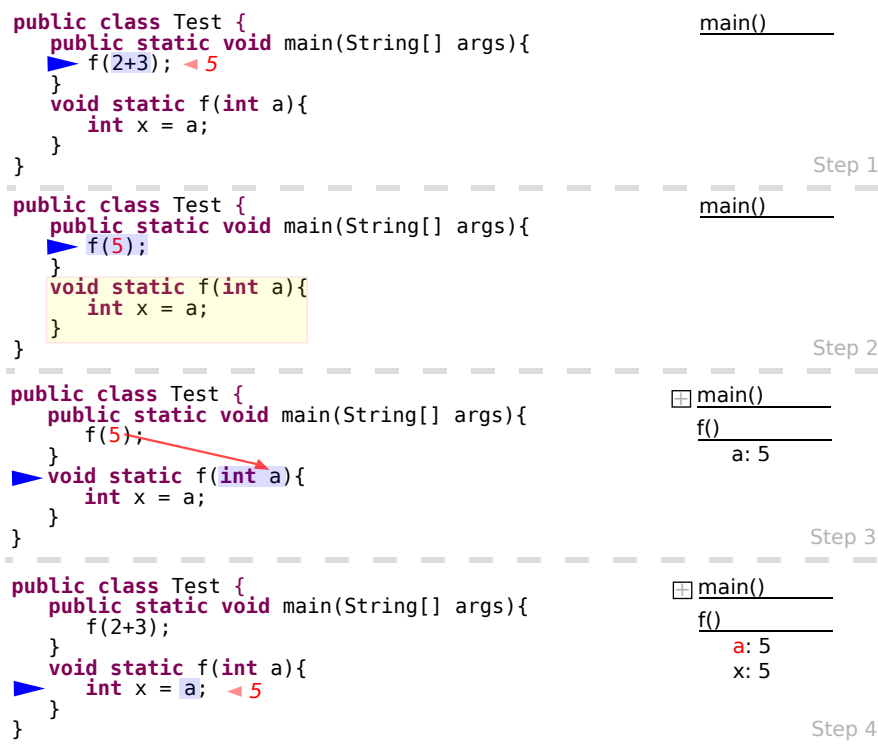


Figure 5.7: Example of parameter passing

BlueJ is an integrated development environment designed for teaching and learning object-oriented programming. The IDE provides a Unified Modelling Language (UML) class diagram that visualise the application structure, direct method calls from the user interface and a breakpoint dependent debugger. We believe that BlueJ’s user interface may be too complicated for many students, as there a many different modules to understand how to use and on top of the primary goal that is learning the Java programming language, they are required to learn the semantics of UML. The debugger only provides a stepwise evaluation of statements, not the entire expression tree as Jarvis provide, and has poor visualisation of the connected scopes.

DrJava is a programming environment for Java that has an integrated Java interpreter (interactions pane). The interpreter provides a *read-eval-print loop* (REPL) interface with Java. Through this interface the user is able to create Java expressions and statements that directly access and use the Java classes, and their members (methods and variables), currently in development (requires compilation).

## 5.6 Conclusion

Javis is targeted at students learning Java as their first programming language, with little or no experience with programming and using complicated software. Understanding why a program segment does not work as intended, and where it fails, is not an easy task for these unexperienced students. We believe that the assistance our tool provides is valuable and that it may reduce the frequency of many of the *concept-based problems*. The availability that a web application features and our focus on usability and simplicity is important for the success of Javis.

## Chapter 6

# Proposal: Grading by Annotation

In this chapter we specify a grading by annotation scheme that we propose as a tool for teaching assistants. We have developed it based on our work in this thesis.

During the research that we performed for this thesis we gained experience and knowledge about what kinds of problems that do occur, and we identified what kinds of situations that require special kinds of annotations. At our department the teaching assistants grade assignments by reading and testing the submitted solutions. The feedback to the students depends on the type of assignment. For exercises, the students are given a pass or a fail, with a short comment that describes any problems. For semester assignments the teaching assistants fill out a form where they for each of the requirements of the assignment give a score between zero and the maximum attainable score, there is also a comment field for each requirement. The detail of the feedback for exercise assignments is quite low, and it may require a lot of text to describe what exactly is wrong and where the problem is located. For semester assignments the feedback may be a lot more specific regarding location, depending on how specific the requirement is formulated.

With our proposed scheme the teaching assistants annotate the exact locations of the problems in the submitted solutions, with annotations that describes the problems. This way the students will have an easier task of connecting the feedback with the problems. It also makes it significantly easier for the teaching assistants to give feedback to the students, and especially regarding lesser problems that with a large workload might be ignored. Though in some situations a comment is necessary to provide adequate feedback, this is also supported by our scheme.

An important feature of this scheme is that the graded solutions can be scanned for annotation occurrences, and a database can be built from

the results. This extracted data can be studied by the teachers or used for research. There are many things that the teachers can learn by studying this data, e.g. they can identify the most common problems of the previous assignment and adjust their lectures and future assignments accordingly.

## 6.1 Annotation Syntax

The syntax we have chosen for our annotation scheme supports a comma separated list of annotations and an optional comment for each location. In addition it is possible to include the optional brace parentheses to mark segments of code that the annotation describes. Listing 6.1 includes all the possible syntax elements, notice the brace closure at line 3. The two optional elements do not depend on each other, and one or the other, both or none can be left out.

In this proposal we have used numbers to represent the identities of annotations, and in the comma separated list of annotations these numbers are inserted for `<anno-id>`. Other syntax schemes can be used as identifiers as well, e.g. named annotations.

Listing 6.1: Syntax of grading by annotation

```
1 ///<anno-id>,...](<comment>){  
2 <code segment>  
3 ///}
```

In Listing 6.2 we have given a list of two annotation identifiers, and have not included any of the optional elements. The semantics of this annotation is that the file has no implementation for any of the assigned requirements, and that the part of the assignment that is this file, is considered as failed. Notice that both normal and brace parentheses are not present. The bare minimum of an annotation is a list with one annotation identifier.

Listing 6.2: Annotation without comment and braces

```
1 ///[3,5]  
2 class Recipe {  
3     ...  
4 }
```



## 6.2 Grading

We have three types of annotations in our scheme, *file annotations*, *problem annotations* and *meta annotations*. A combination of these three should be used when grading and providing feedback to the students. In addition we suggest that the teaching assistants provide comments, using the syntax of our scheme, whenever it is necessary in order to provide the best possible feedback.

The list of annotations in this proposal is not final, and we do recommend that the teaching assistants in collaboration with the lecturer extend it with more annotations when necessary. We emphasise that any addition should be discussed, and that an updated list should be kept available at all times. You will notice that the identifier we have given to the annotations follow a pattern, *file annotations* have numbers ranging from 1 to 7, *meta annotations* from 20 to 28 and each of the different *problem annotation* categories has its own x00-x99 interval. We have chosen to number the annotations in this way to make it easy to differentiate the different annotation categories, and we have chosen a large interval so that it is possible to add additional annotations while keeping the category meaning.

### 6.2.1 File annotations

*File annotations* describes files and their contents, with the primary purpose of stating if the file contains adequately good implementations to pass or not. We have divided the *file annotations* into two types, *grade* and *meta*. In Table 6.1 we have listed the different annotations with a short description for each. Each file of a submitted solution that the student should have implemented something in, must have a *grade* annotation. The file and its contents can be further described using the *meta* annotations listed in the table. We recommend that all these annotations, except for *4-non-implemented-method*, be placed near the top of the file, and preferably on the line above the public class of the file. The *4-non-implemented-method* annotation should, if possible, be placed on the line above any methods it applies to.

The *meta* annotations provide a more detailed description about submitted files, e.g. if some method is not implemented, contains compile errors etc. Combining *grade* and *meta* annotations may often provide valuable information, especially related to research. A good example is *Fail* combined with *File not implemented*, which clearly state that the reason that it was graded with fail is because nothing was implemented, and not some other reason like that it was a bad implementation.

		Grade
1	Pass	The file solves the given problem sufficiently.
2	Almost fail	The file barely solves the given problem, and is almost marked as fail.
3	Fail	The file does not solve the given problem sufficiently.
		Meta
4	Non implemented method	The following method has not been implemented.
5	File not implemented	This file contains no implementations, except for those provided with the assignment.
6	Changed assignment	The structure that was given as a premise with the assignment has been changed. (e.g. the parameter list of a method has been changed)
7	Compile error	The file does not compile.

**Table 6.1:** File annotations

### 6.2.2 Meta annotations

*Meta annotations* are used to further describe the nature of a problem, and must always be connected with a *problem annotation* when used. In Table 6.2 we have provided a list of the annotations that we found necessary in the work we did for this thesis. An example of how *meta annotations* are intended to be used is a situation where a conditional expression is incorrect because the wrong logical connective was used, which should be annotated with *201-incorrect condition* and *22-incorrect logic* (see Table 6.4 and Table 6.2).

### 6.2.3 Problem annotations

*Problem annotations* mark occurrences of mistakes, possible misconceptions, uncovered requirements etc. We have created a list of annotations based on our work in this thesis, and our previous educational experience. They are categorised based on what they relate to:

1. Variables - Table 6.3
2. Conditions - Table 6.4

20	Incorrect arithmetics	Arithmetical expression that is does not solve the problem.
21	Wrong operator	Use an incorrect operator.
22	Incorrect logic	Incorrect because of the logic applied.
23	Missing case	Incorrect because it does not cover all cases.
24	Unnecessary	Unnecessary to provide a correct solution.
25	Breaking	Causes the program to break the solution.
26	Repeating	Repetition that should not be here.
27	Reference/object equality	Used reference equality instead of object equality, or vice versa.
28	Static/instance confusion	Used static access instead of instance access, or vice versa.

**Table 6.2:** Meta annotations

3. Method calls - Table 6.5
4. Scopes - Table 6.6
5. Control structures - Table 6.7

The annotations are intended to describe entire problems, and it should not be necessary to use multiple annotations to describe the same problem. Situations where multiple problems are present will occur, and when grading one should attempt to see if there is a primary problem, and not focus on the minor ones.

Listing 6.3: Multiple problems ✖

```

1 void removeDuplicates(){
2     ///#[204]
3     for(int i = 0; i <= size; i++){
4         int j = i + 1;
5         ///#[508]{
6         ///#[201]
7         if(parts[i] == parts[j]){
8             parts[i].weight += parts[j].weight;
9             remove(j);
10        }
11        ///#}
12    }
13 }
```

101	Unnecessary variable	Declared an unnecessary variable.
102	Incorrect type	Declared an incorrect or less suitable type for a variable.
103	Wrong location	Declared a variable in a wrong, or less suitable, location.
104	Bad name	Declared a variable with a name that does not fit, or badly describes, its purpose.
105	Incorrect assignment	Incorrect assignment to a variable.
106	Should have referred to a variable	The statement or expression contains a sub-expression that is the same as one previously assigned to a variable.

**Table 6.3:** Variable problem annotations

201	Incorrect condition	The conditional expression is incorrect.
202	Almost correct condition	The conditional expression is almost correct.
203	Unnecessary condition	The conditional expression, or sub-expression, is unnecessary.
204	Boundary case condition	Incorrect boundary case expression. (e.g. less-than-or-equal to operated instead of strictly-less-than).

**Table 6.4:** Conditional expression problem annotations

Listing 6.3 is such a situation where multiple problems are present, but there is one larger problem that overshadows the others. The example is extracted from a student submitted solution, that was inspected in the research we presented previously in this thesis. A short description and a class diagram of the assignment is given in Section 4.5. The requirement given in the assignment of the method `removeDuplicates()` is that it should combine and remove any duplicate `RecipePart` objects in the list parts, i.e. if they have the same ingredient then their weight should be added and one of the parts removed. The submitted solution contains multiple problems, incorrect boundary case condition on line 3, incorrect condition on line 7 and most important, a missing nested loop-construct. It is likely that the student did not understand how to nest loop-constructs to remove all the duplicates, and that the presence of the two incorrect conditions is a consequence of that. We recommend that the teaching assistants discuss their choices during the semester, and attempt to find a common reasoning.

301	Repeated method call	This method call is unnecessarily repeated.
302	Unused return value	The returned value of this method call, to an observer-method, is not assigned to a variable, and does not affect the outcome of the program.
303	Incorrect method call	The method call is incorrect or incorrectly specified. (e.g. it is a call to the wrong method, contains incorrect arguments etc.)
304	Missing method call	There should be a method call at this location.

**Table 6.5:** Method call problem annotations

### 6.3 Collecting the Data

One of the most important reasons to choose this grading scheme is to be able to study what problems the students have, how frequent they are, correlations between the problems etc. To do this the graded solutions must be scanned for occurrences of annotations, and the results gathered in a database. In the study we performed in this thesis we saved additional data with the annotation identifier:

- Assignment identifier.
- Student identifier (Anonymous if necessary).
- Filename (requires that all submitted solutions have the exact same name for all the files).
- Line number of mistake.

This level of detail allowed us to analyse the data in many interesting ways, see Chapter 4, and there are many more possibilities.

To be able to collect all these details we kept a folder structure, with one folder for each assignment that contained one folder for each student. Then we developed a simple BASH script that scanned the files for annotations, and through the folder structure created SQL insert statements with all the details listed above. Additionally we suggest that the tool checks that the scanned files adheres to the specified rules by checking that:

- the file has a pass or fail annotation.
- that any file with the fail annotation also has at least one other annotation, to guarantee feedback to the student.

401	Wrong variable	Referring to a variable from the wrong namespace or scope, e.g. assignment to a formal parameter instead of a field variable etc.
402	Global variable	Declared a global variable to pass a value across namespaces and/or scopes.
403	Non-existing variable	Referring to a variable not available in the current scope, e.g. undeclared, declared in a child scope, declared in the calling scope, formal parameter of a method in the method call's scope etc.
404	Parameter passing	Misconception of parameter passing semantics, e.g. referring to a variable passed as an argument instead of the formal parameter it was passed to, declaring a variable with the same name as a formal parameter etc.

**Table 6.6:** Scope problem annotations

- that mutually exclusive annotations do not occur together.
- that all annotations that are used, are declared.

501	Complicated structure	The following control structure is overly complicated, difficult to understand or does not resemble a solution that would solve the assigned problem etc.
502	Incorrect grouping	Incorrect placement of braces, incorrectly nested structures, incorrectly connected structures (if-else) etc.
503	Missing if-sentence	The following program segment should be guarded by an if-sentence.
504	Breaking if-sentence	The following program segment should not be guarded by an if-sentence.
505	Loop/if confusion	The following program segment has a loop-construct instead of an if-sentence, or vice versa.
506	Manually solved a loop problem	The following manually solved program segment should be solved using a loop-construct.
507	Incorrect use of a loop	The following program segment should not be solved using a loop-construct.
508	Loop instantiation problem	The following problem should be solved with a loop, and it is either missing or incorrectly instantiated.
509	Missing statement or structure	There should be control statement or structure here.

**Table 6.7:** Control structure problem annotations





# Chapter 7

## Conclusion

### 7.1 Status Summary

The primary goals for this thesis were:

1. Develop a Java visualisation tool to assist students in their learning process.
2. Study concrete student problems, in order to be able to target the tool 1 at real-world problems faced by students.
3. Design a grading by annotation scheme, in order to support 2 and as an aid in future grading.

We reached goal 2 by performing an empirical study, where we inspected 349 student submitted solutions, and annotated any problems we found. The annotated solutions were scanned and the results were inserted into a database for study. Among the students who submitted a solution that we inspected, we found that:

- 60% struggled with method calls in general.
- 47% had solutions that were missing necessary method calls in their solutions.
- 82% had troubles related to loop constructs.
- 93% had incorrect conditional expressions.
- contrary to our initial hypothesis, none had mistakes related to references.

- 20% had solutions that had mistakes related to parameter passing.

We defined a problem taxonomy where problems are classified as *knowledge-based* or *concept-based* by looking at the plausible underlying reasons for the problems. *Knowledge-based* problems are those that can be related to lack of knowledge or understanding, and *concept-based* problems are those that can be related to observable misconceptions of constructs and how they affect the computer. By classifying the problems that we found were the most common, we could conclude that a significant part of the students' problems were *concept-based*, hence observable. We argue that with the right aid while programming, the students can be able to find and solve these problems themselves.

Based on the experience and knowledge we gained during the work for goal 2 we developed a grading by annotation scheme, and we consider goal 3 as reached. It is designed to be used in introductory programming courses as the primary channel for feedback to the students, and as a tool to collect data about the problems the students have. Through this scheme the teachers can collect valuable feedback of how their lectures affect the students' performance on assignments and if the assignments have the correct focus.

Goal 1 is not finished, but many of the sub goals are reached and we plan to reach most of the other sub goals before the WCRE conference of 2013<sup>1</sup>. We do have an implementation that can evaluate all the basic language features, except for loop constructs. It is primarily visualisation features that are missing.

## 7.2 Future Work

In this section we briefly explore the possibilities and future work of both *Javis* and the *grading by annotation* scheme.

### 7.2.1 Javis

The tool and the AST transformation web service are both at a prototype stage. In this section, we describe the future work for these components.

- AST transformation web service.
  - Restructure as a servlet [22] that can be deployed in a Java Servlet container, e.g. Tomcat [3], GlassFish [19].

---

<sup>1</sup><http://wcre.wikidot.com/2013>

- Support transformation of all Java language constructs.
  - Improve documentation.
  - Handle compiler errors.
- Javis.
  - Implement all the specified requirements.
  - Improve documentation.
  - Improve compliance with the Java Language Specification [21].
  - Add support for editing submitted source code.
  - Add support for assigning values to variables of running programs.
  - Implement visualisation of references.

We encourage others to study if there are advantages of using Javis as an aid while learning Java.

### **7.2.2 Grading by annotation**

We believe that our scheme is ready to be used in a course. Different institutions have different ways of teaching introductory programming, and we recommend any users of the scheme to adapt, refine and improve it to fit with the requirements of their situation. During the semester, when the grading of an assignment is complete, this scheme provides the educators with the possibility to study the problems that the students had, and we recommend that this advantage is used actively. The educators can analyse the problems of an assignment, and then adapt the following lectures and assignments to focus on these challenges – actively checking if this improves the situation.

We also encourage research into student problems using the data made available through the use of this scheme. It would be interesting to study the results of our study in relation to other semesters of INF100 at our department, and equivalent courses at other institutions as well. There may also exist interesting correlations that can be studied pertaining student problems, we were unfortunately not able to do this, since our data set is quite small.

















# Appendix A

## Student problems

ID	Name	Description
↘ <sup>PASS</sup>	Pass	File that solves the given problem, or displays sufficient understanding of how.
↘ <sup>FAIL</sup>	Fail	File that does not solve the given problem, nor display sufficient understanding of how.
↘ <sup>POOR</sup> ↘ <sup>UNDR</sup>	Poor understanding	Locations or files where the student displays a poor understanding of either the problem or the code written.
↘ <sup>CHNG</sup> ↘ <sup>EXER</sup>	Changed the assignment	The student has changed (or removed) parts of pre-defined code for the assignment, i.e. removing methods to in-line them, changed parameter list etc.
↘ <sup>UNSOLV</sup> ↘ <sup>PROB</sup>	Unsolved problem part	Missing implementation of a method, or large segment, of the assigned problem.
↘ <sup>COMP</sup> ↘ <sup>ERR</sup>	Compile error	File that does not compile.
↘ <sup>NOT</sup> ↘ <sup>IMPL</sup>	Not implemented	File that is not implemented at all.
↘ <sup>HARD</sup> ↘ <sup>CODE</sup>	Hard-coded solution	Part of a problem solved by circumventing the specified way to solve the problem by writing a fixed solution.
↘ <sup>SYN</sup> ↘ <sup>ERR</sup>	Syntax error	File or location with syntax error.
↘ <sup>BAD</sup> ↘ <sup>OP</sup>	Wrong/non-existing operator	Usage of a non-existing or wrong operator for that problem.

ID	Name	Description
<del>7</del> <sup>XTRA</sup> CALL	Repeated method calls	Unnecessary repeated method calls, first without using the return value, followed by another call where the return value is used (e.g. assigned).
<del>7</del> <sup>UNUSD</sup> VAL-S	Unused return value (static)	Calls a static method without using the returned value.
<del>7</del> <sup>UNUSD</sup> VAL-O	Unused return value (object)	Calls a method on an immutable object (e.g. String) without using the return value.
<del>7</del> <sup>BAD</sup> ARIT	Erroneous arithmetic	Arithmetic that does not solve the given problem.
<del>7</del> <sup>STATIC</sup> CALL	Instance method in a static way	Attempted to call an instance method in a static way, with the object in question as an additional argument.
<del>7</del> <sup>NO</sup> RET	Missing return statement	Method without return statements at all possible paths.
<del>7</del> <sup>NO</sup> CALL	Missing method call(s)	Lack of understanding of which method calls that should have been called at that location.
<del>7</del> <sup>BAD</sup> CALL	Incorrectly specified method call	Lack of understanding how to correctly specify a method call.
<del>7</del> <sup>TYPE</sup> PROM	Error due to type promotion	Lack of understanding how automatic type promotion affects the results of an expression.
<b>Variables</b>		
<del>7</del> <sup>XTRA</sup> VAR	Unnecessary	Declared unnecessary variable(s)
<del>7</del> <sup>BAD</sup> TYPE	Less suitable type	Declared a variable with a less suitable type, e.g. double to hold an integer value.
<del>7</del> <sup>UNUSD</sup> VAR	Unused	Declared unused variable(s)
<del>7</del> <sup>PRE</sup> DECL	Pre-declared	Unnecessarily pre-declared variables.
<del>7</del> <sup>XTRA</sup> ASSGN	Unnecessary assignment	Unnecessary assignment to a variable.
<del>7</del> <sup>LOOP</sup> CNTRS	Multiple loop counters	Declared multiple loop counter variables for non-nested loops in the same scope.
<del>7</del> <sup>UGLY</sup> ASSGN	Breaking assignment	Assignment that breaks the intended functionality.

ID	Name	Description
 SHORT NAME	One-letter variable names	Using many variables with one-letter names.
 RPT EXPR	Restating expression stored in variable	Copy-paste of expression assigned to a variable instead of referring to the variable
 WRNG ASSGN	Assignment to local variable	Assigned a value to a local variable with the intention of affecting a variable in the calling scope, e.g. assigning a value to a parameter variable with the same name as a variable in the calling scope.
 GLBL VARS	Global variables	Declaring global static variables to access values from the calling scope instead of using the intended parameter transmission.
 WRNG SCOPE-I	Inner scope variables	Refers to variables from an inner scope
 ARRAY SNTX	Primitive/array confusion	Refers to a variable using array syntax or to an array without the correct syntax.
 CALL SNTX	Variable/method confusion	Refers to a variable using method call syntax.
 WRNG SCOPE-S	Sum variable	Declares the sum variable in the wrong scope, i.e. declaring it inside the scope of the loop iterating over an array.
 WRNG TYPE	Wrong type	Declared a variable of the wrong type.
 WRNG ASSGN	Assignment	Lack of understanding of what value to assign to a variable
 ARG SCOPE	Calling scope variable	Refers (illegally) to the variable from the calling scope that was used as an argument, instead of referring to the formal parameter variable.
 UNDECL VAR	Undeclared variable	Refers to undeclared variable.
 PARAM PASS	Same name as formal parameter	Declared a local variable with the same name as a formal parameter variable of the method called in one of the following lines.
 FRML PARAM	Non-existing formal parameter	Attempts to pass a non-existing variable with the same name as a formal parameter variable of the method being called as an argument, instead of the existing variable.

ID	Name	Description
123 7 SELF ASSGN	Assignment of self	Assigns a variable or array position to itself, or an expression that was just assigned.
<b>Control structures</b>		
202 7 CMLX CTRLS	Unnecessarily complicated	Using control structures in a very complicated way.
203 7 IF/LOOP CONF	Loop/if confusion	Used a loop instead of an if-sentence, or vice versa.
204 7 WRNG GRPNG	Incorrect grouping	Incorrect placement of braces, incorrectly nested, incorrect if-else combination.
206 7 MAN LOOP	Manual loop problem	Manually solved a loop problem, in most cases this also counts as a hard-coded solution.
207 7 BDRY COND	Boundary case condition	Incorrect boundary case condition, applies to conditions of both if-sentences and loops.
208 7 WRNG LOOP	Incorrect use of a loop	Attempted to solve a problem with a loop, that has to be solved in another way.
<b>If statements</b>		
301 7 XTRA COND	Unnecessary condition	A condition or if-sentence that does not affect the results of the solution.
302 7 UGLY IF	Breaking sentence	An if-sentence that breaks the behaviour of the solution.
303 7 WRNG COND	Incorrect condition	An incorrect condition that breaks the solution.
304 7 BOOL EQL	Boolean equality	Conditional expression that is checked for equality with a boolean literal.
305 7 ALMST CRRCT	Almost correct	Some parts of the conditional expression is correct.
306 7 BOOL ACCUM	Accumulate boolean	Does not accumulate the boolean value as the problem description specifies, i.e. check if all parts of a loop was successful.
<b>Loop statements</b>		
401 7 UNSCRY LOOP	Unnecessary loop	Used a loop where it should not be used, without breaking the results.
402 7 LOOP CREATE	Loop instantiation problem	The student has not understood how to solve the problem using a loop, and left it unsolved.



# Appendix B

## Annotation occurrences

Annotation	Total occurrences	Students	Files	Students repeated
1/ PASS	808	79	808	75
2/ FAIL	94	43	94	16
3/ POOR UNDR	26	16	22	2
4/ CHNG EXER	9	9	9	0
5/ UNSOLV PROB	103	46	83	18
6/ COMP ERR	79	33	79	11
7/ NOT IMPL	20	14	20	3
8/ HARD CODE	31	16	29	2
9/ SYN ERR	46	15	22	3
21/ BAD OP	7	7	7	0
22/ XTRA CALL	9	8	8	0
23/ UNUSD VAL-S	4	4	4	0
24/ UNUSD VAL-O	8	7	8	0
25/ BAD ARIT	45	29	43	9
26/ STATIC CALL	1	1	1	0
27/ NO RET	5	4	4	0
28/ NO CALL	75	37	52	7
29/ BAD CALL	35	17	23	4
30/ TYPE PROM	12	11	11	0
101/ XTRA VAR	81	43	76	14
102/ BAD TYPE	10	8	9	0
103/ UNUSD VAR	3	3	3	0
104/ PRE DECL	99	44	97	17
105/ XTRA ASSGN	10	10	10	0

Annotation	Total occurrences	Students	Files	Students repeated
106 LOOP CNTRS	13	13	13	0
107 UGLY ASSGN	20	14	16	1
109 SHORT NAME	25	20	24	0
110 RPT EXPR	4	4	4	0
111 WRNG ASSGN	20	12	14	2
112 GBLB VARS	1	1	1	0
113 WRNG SCOPE-I	4	3	3	0
114 ARRAY SNTX	8	5	7	1
115 CALL SNTX	2	2	2	0
116 WRNG SCOPE-S	1	1	1	0
117 WRNG TYPE	5	5	5	0
118 WRNG ASSGN	8	5	7	1
119 ARG SCOPE	1	1	1	0
120 UNDCB VAR	7	5	5	0
121 PARAM PASS	27	22	27	4
122 FRML PARAM	2	2	2	0
123 SELF ASSGN	11	9	10	0
202 CMLX CTRLS	53	40	48	2
203 IF/LOOP CONF	7	4	4	0
204 WRNG GRPNG	39	30	36	3
206 MAN LOOP	27	16	27	0
207 BDRY COND	119	58	100	28
208 WRNG LOOP	7	6	6	0
301 XTRA COND	14	11	12	1
302 UGLY IF	3	3	3	0
303 WRNG COND	113	55	80	17
304 BOOL EQL	69	39	49	8
305 ALMST CRRCT	16	15	16	1
306 BOOL ACCUM	44	29	30	0
401 UNSCRY LOOP	28	19	22	3
402 LOOP CREATE	72	42	62	11

# Bibliography

- [1] E. Allen, R. Cartwright, and B. Stoler. DrJava: a lightweight pedagogic environment for Java. *SIGCSE Bull.*, 34(1):137–141, Feb. 2002.
- [2] J. Anderson. *The Architecture of Cognition*. Cognitive science series. Lawrence Erlbaum Associates, 1996.
- [3] Apache Software Foundation. Apache Tomcat, 2013.
- [4] F. Bailie, M. Courtney, K. Murray, R. Schiaffino, and S. Tuohy. Objects first - does it work? *J. Comput. Sci. Coll.*, 19(2):303–305, Dec. 2003.
- [5] P. Bayman and R. E. Mayer. A diagnosis of beginning programmers' misconceptions of basic programming statements. *Commun. ACM*, 26(9):677–679, Sept. 1983.
- [6] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.
- [7] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). Technical Report RFC 4627, Network Working Group, 2006.
- [8] N. Dale. Content and Emphasis in CS1. *The SIGCSE Bulletin*, 37(4):69–73, 2005.
- [9] P. J. Denning and A. McGettrick. Recentering computer science. *Commun. ACM*, 48(11):15–19, Nov. 2005.
- [10] B. du Boulay. Some difficulties of learning to program. In E. Soloway and J. Spohrer, editors, *Studying the novice programmer*, pages 283–299, 1989.
- [11] Google, Inc. The Dart programming language, July 2013.

- [12] W. L. Johnson, E. Soloway, B. Cutler, and S. Draper. Bug catalogue: I. Technical Report 286, Department of Computer Science, Yale University, New Haven, CT, 1983.
- [13] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ system and its pedagogy. *Computer Science Education*, 13(4):249–268, 2003.
- [14] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen. A study of the difficulties of novice programmers. *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education - ITiCSE '05*, page 14, 2005.
- [15] M. Linn and J. Dalbey. Cognitive consequences of programming instruction. In E. Soloway and J. Spohrer, editors, *Studying the novice programmer*, pages 57–81, 1989.
- [16] L. Ma, J. Ferguson, M. Roper, and M. Wood. Investigating the viability of mental models held by novice programmers. *SIGCSE Bull.*, 39(1):499–503, Mar. 2007.
- [17] L. Ma, J. D. Ferguson, M. Roper, I. Ross, and M. Wood. Using cognitive conflict and visualisation to improve mental models held by novice programmers. *SIGCSE Bull.*, 40(1):342–346, Mar. 2008.
- [18] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. *SIGCSE Bull.*, 33(4):125–180, Dec. 2001.
- [19] Oracle, Inc. GlassFish, 2013.
- [20] Oracle, Inc. Java Compiler Tree API, 2013.
- [21] Oracle, Inc. Java Language Specification, 2013.
- [22] Oracle, Inc. Java Servlet Technology, 2013.
- [23] D. N. Perkins, C. Hancock, R. Hobbs, F. Martin, and R. Simmons. Conditions of learning in novice programmers. *Journal of Educational Computing Research*, 2(1):37–55, 1986.
- [24] A. Robins, J. Rountree, and N. Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, pages 37–41, April 2003.

- [25] C. Schulte and J. Bennedsen. What do teachers teach in introductory programming? *Proceedings of the 2006 international workshop on Computing education research - ICER '06*, page 17, 2006.
- [26] J. Spohrer, E. Soloway, and E. Pope. A goal/plan analysis of buggy Pascal programs. In E. Soloway and J. Spohrer, editors, *Studying the novice programmer*, pages 355–399, 1989.
- [27] J. C. Spohrer and E. Soloway. Novice mistakes: are the folk wisdoms correct? *Communications of the ACM*, 29(7):624–632, July 1986.