

BACHELOR THESIS  
COMPUTING SCIENCE



RADBOD UNIVERSITY

---

**Designing and evaluating a  
learning activity about debugging  
for novice programmers**

---

*Author:*  
Ruben Holubek  
s1006591

*First supervisor/assessor:*  
Sjaak Smetsers  
S.Smetsers@cs.ru.nl

*Second assessor:*  
Erik Barendsen  
Erik.Barendsen@ru.nl

June 25, 2020

## **Abstract**

Debugging is a very important skill for a programmer and therefore, it is useful if this skill is explicitly taught. However, debugging is often an underrepresented topic in education and is overshadowed by other subjects. The main reason for this is that teachers do not have the time and teaching material to explicitly teach debugging to their students. To tackle this problem, we designed an effective learning activity about debugging.

In our designed learning activity, we taught the systematic debugging procedure designed by Michaeli and Romeike (2019) and several debugging strategies which were useful and practical for the students, namely using JavaDoc, systematically evaluating a run time error and using the debugger. Knowing and applying these debugging procedure and strategies were the learning goals of this lecture. The principles of direct instruction were also integrated into this lecture, which were achieved by having several demos, small exercises between different subjects and a debugging exercise to practice with afterwards. This led to a highly interactive lecture.

After the lecture, we interviewed several students twice to evaluate the learning activity. In these interviews, students solved a debugging exercise and were asked some questions. From these evaluations and the pre-test became clear that all the learning goals were achieved and therefore, that the learning activity was effective.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Debugging . . . . .	4
1.2	The problems regarding debugging in education . . . . .	5
1.2.1	Students' unfamiliarity regarding debugging . . . . .	5
1.2.2	Problems with teaching debugging . . . . .	5
1.3	Current knowledge . . . . .	6
1.4	Overview of this research project . . . . .	6
1.5	Structure of this thesis . . . . .	7
<b>2</b>	<b>Background</b>	<b>8</b>
2.1	Different types of errors . . . . .	8
2.2	Encountered problems regarding debugging . . . . .	9
2.3	Teaching a systematic debugging procedure . . . . .	10
2.3.1	Michaeli and Romeike's approach: a demo . . . . .	10
2.3.2	Allwood and Björhag's approach: a document . . . . .	10
2.3.3	Other approaches . . . . .	11
2.4	Effectively teaching new material: direct instruction . . . . .	11
2.4.1	Applying direct instruction . . . . .	12
2.4.2	Benefits of direct instruction . . . . .	13
2.5	Measuring the effect of an educational intervention . . . . .	13
<b>3</b>	<b>Goal</b>	<b>15</b>
<b>4</b>	<b>Methodology: Design</b>	<b>16</b>
<b>5</b>	<b>Results: Design</b>	<b>19</b>
<b>6</b>	<b>Methodology: Evaluation</b>	<b>23</b>
6.1	Pre-test . . . . .	24
6.1.1	Data collection . . . . .	24
6.1.2	Data analysis . . . . .	24
6.2	First interviews . . . . .	24
6.2.1	Data collection . . . . .	24
6.2.2	Data analysis . . . . .	28

6.3	Second interviews . . . . .	30
6.3.1	Data collection . . . . .	30
6.3.2	Data analysis . . . . .	31
6.4	Combining the results . . . . .	31
<b>7</b>	<b>Results: Evaluation</b>	<b>32</b>
7.1	Pre-test . . . . .	32
7.2	First interviews . . . . .	32
7.2.1	Achievement of the learning goals . . . . .	33
7.2.2	Pros and improvements . . . . .	33
7.2.3	Another interesting observation . . . . .	35
7.3	Second interviews . . . . .	36
7.4	Combining the results . . . . .	37
<b>8</b>	<b>Conclusions</b>	<b>38</b>
8.1	Research subquestions . . . . .	38
8.2	Research question . . . . .	39
<b>9</b>	<b>Discussion</b>	<b>41</b>
9.1	Reflection on findings . . . . .	41
9.2	Reflection on methods . . . . .	42
9.3	Implications for practice . . . . .	43
9.4	Future research . . . . .	44
	<b>References</b>	<b>45</b>
<b>A</b>	<b>Figures</b>	<b>47</b>
<b>B</b>	<b>Code Fragments</b>	<b>51</b>
B.1	Debugging exercise after presentation: Calculate average price of a list of fruits . . . . .	52
B.2	Debugging exercise of the interview: Calculate median of a list of integers . . . . .	54
B.3	JavaDoc demo used in the presentation . . . . .	55
B.4	Run Time error demo used in the presentation . . . . .	56
B.5	The first debugger demo used in the presentation . . . . .	56
B.6	The second debugger demo used in the presentation . . . . .	57
<b>C</b>	<b>Documents</b>	<b>58</b>
C.1	Pdf answer debugging exercise after presentation . . . . .	59
C.2	Pdf answer debugging exercise used in the interview . . . . .	60
C.3	Schema used for the interviews . . . . .	61
C.4	Socrative Quiz used in the presentation . . . . .	62
C.5	Filled in interview schemes . . . . .	64
C.6	Results of the second interviews . . . . .	74

C.7	Used slides in the presentation . . . . .	75
-----	---	----

# Chapter 1

## Introduction

### 1.1 Debugging

Programmers often make mistakes when writing a program, which can be caused by a simple typo, a forgotten edge case or a logical fault. These bugs result in the program behaving differently than expected and so, these bugs should be found and fixed and this process is called debugging. Thus, debugging is the process of identifying and removing errors in programs.

Debugging is actually an often overlooked, but rather important skill of a programmer, because programmers spend a lot of time debugging their programs instead of expanding them; senior developers spend around 30% of their time debugging programs (Perscheid, Siegmund, Taeumel, & Hirschfeld, 2017). Therefore, being able to effectively debug your program is a valuable asset for every programmer, because that could shorten the duration of the development of a program. To make the debugging process as fast and easy as possible, this process is often supported by different systems that are available in the IDE or other programs. The most known example is the debugger, but available documentation and error highlighting in IDE's are also very helpful. But these systems and debugging strategies are not that useful if they are not correctly used at the right moments, e.g. compile time errors are not the bugs that are found with the debugger. Therefore, it is important to use an effective debugging procedure which is supported by the different strategies and systems provided in the IDE. It is especially really useful to know a systematic debugging process to systematically fix the bug in the program to find the bugs as quickly as possible and to shorten the time spend on debugging.

This is even more relevant to programming novices. They are learning to program and they often make more errors when writing their programs, because they do not have necessary experience yet. If programming novices learn to effectively debug their programs early on, they will be able to save a lot of time solving their bugs; not only in class, but also later when they

might be a senior developer. Therefore, the process of learning to program could be facilitated by an effective debugging process and other debugging strategies (Michaeli & Romeike, 2019).

## **1.2 The problems regarding debugging in education**

### **1.2.1 Students' unfamiliarity regarding debugging**

Contradictory, many students are not familiar with such a debugging process or debugging strategies in general. They are unfamiliar with the debugger and do not know the strategies used to debug a program efficiently and as a result, they apply a trial-and-error technique. This technique consists of placing print statements between the different lines of code to see the flow of the different variables. This method is not very effective to debug a program and mostly relies on luck, because most print statements are not placed in an effective, but rather unstructured manner. Moreover, in many debugging cases, several variables are relevant to the problem. To keep an eye out to these variables, even more print statements are needed to inspect the flow of the program. Not only does this disturb the readability of the code, but when the bug is fixed, all the print statements should be removed, which takes some time and if several statements are forgotten, the code is still scattered with unnecessary statements. This strategy takes a lot of unnecessary time and as a result, students feel helpless and frustrated when they cannot find the bug in their program (Michaeli & Romeike, 2019).

### **1.2.2 Problems with teaching debugging**

Teaching students an effective, systematic debugging process and other debugging strategies when they are learning to program would solve this problem. So, why are students not learning debugging strategies to effectively debug their programs and to improve their programming skills as well? This problem lies in the fact that debugging is an underrepresented topic in class, because teachers often spend their time on other important programming concepts, even though debugging is also an important topic. A reason for this underrepresentation is that debugging is not an explicit content of the curriculum. As a result, it is often overshadowed by other concepts which are actually mentioned in the curriculum (Michaeli & Romeike, 2019). Another reason, and the main reason as indicated by teachers, is the lack of time to teach students the needed debugging skills, but not only in lessons. They often do not have the time to prepare such a lesson, because they lack teaching material. This material includes the suitable concepts, debugging exercises and slides for example (Michaeli & Romeike, 2019). These are the main reasons that debugging is often not included in the programming

courses and as a consequence, the students are not familiar enough with debugging concepts and strategies, which could potentially improve their debugging skills.

### 1.3 Current knowledge

Several researchers already investigated issues related to this problem. Some papers looked into the most frequently occurring problems students encounter when they are programming or when they are debugging. Others inspected the effects of explicitly teaching students a systematic debugging procedure and debugging strategies. Most of these researches indicated several benefits and therefore showed that actually teaching students debugging strategies is useful and really necessary. Papers related to this are inspected in the next chapter.

However, no researchers actually created concrete teaching material that teachers could use in their course program and tested if it was effective and how it could potentially be improved. If such material actually provides the students the benefits of debugging and helps them with the most frequently occurring problems, it would at least tackle the aforementioned problems or maybe even solve them, because other teachers could use this material or improve it.

### 1.4 Overview of this research project

We will design a learning activity to provide some teaching material about debugging. We will also evaluate this learning activity to see how it affects the debugging skills of the students and, based on these and other findings, recommend improvements. Students of the Bachelor's program Computing Science at the Radboud University have several programming courses early on, but none of these courses teach an efficient debugging strategy. Therefore, this is a perfect environment to design and evaluate such a learning activity about debugging. Next to a presentation on relevant debugging concepts, a debugging exercise is provided such that the students can immediately practice with the taught principles.

Students will be interviewed afterwards to see the effects of the learning activity. These interviews consist among other things of a debugging exercise and questions to see how the students tackle these problems immediately after the learning activity in a debugging exercise, but also later in practice. Moreover, we asked them what the good points of the lecture were and potential improvements. Combined with a pre-test, we could research what they learned from the designed lecture. With these results, the created material will be evaluated and can be improved on our recommendations. This should provide the fundamentals of teaching material and a learning



activity that other universities and schools can use as well to ultimately solve the aforementioned problems.

## **1.5 Structure of this thesis**

This thesis is structured as follows: in chapter 2, several studies are discussed to get a better insight in the current problems regarding debugging in education, the approaches of similar researches, methods for effectively teaching new material and other relevant topics. In chapter 3, the answered research questions and subquestions are presented. Afterwards, our thesis is split into 2 parts; a design and an evaluation part. In chapter 4, the steps for structurally designing the learning activity are presented and elaborated. In chapter 5, the results of the performed steps are presented and thus the concrete learning activity is shown and explained. In chapter 6, the methods for evaluating the designed learning activity are presented and elaborated. In chapter 7, the results of these evaluations are discussed and analysed. In chapter 8, the answers to the research (sub)questions are given and conclusions are drawn. In chapter 9, several influencing factors and ideas for future research are presented in the discussion. At the end will be several appendices, which will include among other things the used debugging exercises and the presentation used in the learning activity.

## Chapter 2

# Background

### 2.1 Different types of errors

Debugging is all about solving errors and so it is useful to have a clear terminology for the different types of errors. These terms are used in many papers and will be used in this research as well.

There are 2 different distinctions of errors which both cover all the different types of errors:

- **Compile time, run time and logic errors**

- Compile time errors occur when the written program cannot be compiled. These are already often indicated by the used IDE, but when an error message is given, the line with the bug is indicated.
- Run time errors occur when the program is syntactically correct, but something goes wrong when the program is executing, e.g. a division by 0. These error messages also give an indication of the line where the error occurred.
- Logic errors occur when the execution of the program gives no errors, but the program does something different than the programmer actually meant. No error message is provided for these errors and therefore, logic errors are the most difficult to actually debug.

- **Syntax, type, semantic and logical errors**

- Syntax errors occur when the written code is not grammatically correct, e.g. a missing bracket.
- Type errors occur when the indicated types are not correct, e.g. putting the integer 4 in a boolean variable. Not all languages check on types, so these errors don't occur in all programming languages.

- Semantic errors is a bit ambiguous and many programmers use a different definition. The definition used in this paper will be the definition given by Allwood and Björhag (1991): Semantic errors are syntactically correct expressions which are impossible in the context they occur in. This could be using a variable or function that is not earlier defined; this is syntactically correct, but cannot be executed in the program.
- Logical errors are errors where the code is syntactically correct but does something different than is stated in the problem instruction (Allwood & Björhag, 1991).

One interesting point is that logic errors are not the same as logical errors; a division by 0 will be a logical error, but won't be a logic error, but a run time error instead. Roughly said, syntax, type and semantic errors are covered by compile time errors and logical errors are covered by run time and logic errors, but this also depends on the language used.

## 2.2 Encountered problems regarding debugging

Debugging is an essential skill for every programmer, but it is still under-represented in the classroom. A reason for this underrepresentation is that debugging is not an explicit content of the curriculum and is often overshadowed by other concepts which are actually mentioned in the curriculum as a result (Michaeli & Romeike, 2019). Another reason for this is that there does not exist enough learning material for explicitly teaching debugging and teachers are too busy with explaining other concepts which are seen as more important than debugging (Michaeli & Romeike, 2019). As a result, students are not familiar with an effective systematic debugging method and will use a lot of print statements instead to debug their program. This technique relies on luck and will clutter their code which leads to a long and frustrating debugging process (Michaeli & Romeike, 2019). Students know that the debugger exists, but using it is seen as difficult and complex and that is the reason why they won't use it (Böttcher, Thurner, Schlierkamp, & Zehetmeier, 2016). The biggest struggle encountered by students while debugging is locating the lines where the mistake is made; once the location is found, students are able to fix the bug rather quickly (Katz & Anderson, 1989). This can be explained by the fact that the students are not using a systematic debugging process and no debugger.

A study of Katz and Anderson (1989) researched the most frequently made errors in Java by students and concluded that the most frequent errors are syntax errors. However, these errors are also solved easily and are therefore not so interesting; more interesting are the logical errors and the type errors. These errors occur the most after the syntax errors and should be focused on when providing debugging tips.

## 2.3 Teaching a systematic debugging procedure

Several studies tried to tackle this problem and most of them performed an educational intervention that taught the students a systematic debugging procedure. The main conclusions of these studies were that teaching such a method had several positive impacts on the students and solved the aforementioned problems.

### 2.3.1 Michaeli and Romeike’s approach: a demo

In April 2019, Michaeli and Romeike analysed 12 German high-school teachers’ interviews to investigate, among other things, which debugging skills are conveyed in class and why certain debugging skills are taught or not taught. The main outcomes were that teachers do not explicitly teach debugging, because teachers lack a systematic approach for debugging and there is not enough teaching material. As a result, debugging is an underrepresented topic in class and students use an ineffective trial-and-error which leads to frustration.

To tackle these stated problems, Michaeli and Romeike developed an intervention in October where they taught a systematic debugging process and examined the effects afterwards (Michaeli & Romeike, 2019). They designed a systematic debugging procedure which was based on the ”scientific method”: based on the observed behaviour of the program, repeated hypotheses are formulated, verified in experiments and, if necessary, refined until the cause is found (Zeller, 2009). The procedure is schematically represented in figure A.1. The process distinguishes between different phases which correspond with the different error types (Compile, Run and Compare), because the debugging method depends on the actual error type. Therefore the cycle per error is different, but they share the same steps in general: read and understand the error message (if applicable), identify the cause and modify/make an assumption, determine the error and find the relevant line(s) of code and finally, adjust the program. The undoing of changes is also explicitly mentioned to prevent that students add more errors instead of solving errors.

The main conclusions were that this intervention and the provided debugging had a positive impact on debugging self-efficacy and lead to a better performance on the debugging exercises. However, Michaeli and Romeike suggest that the cycles in their method should be extended with concrete tips and strategies, because these loops are currently relatively abstract.

### 2.3.2 Allwood and Björhag’s approach: a document

Allwood and Björhag (1991) created a document that included several instructions which can be seen as a systematic debugging method and a few

examples of common errors with tips on how to handle them. The instructions that were given can be found in figure A.2, but they come down to interpreting the error message and locating the bug with the given information. If it is a syntax error, it should be fixed relatively easily, but if it is a different error, the flow of information in the program should be carefully inspected. Students were asked to think-aloud while inspecting this flow, but it was unclear if this actually had a positive impact on the debugging process. Finally, after correcting a bug, students were asked to explicitly think about the consequences of the change, so that there would not be unintended consequences. After reading this document, several practice exercises were provided to the students.

The most important conclusions of this study were that teaching a systematic debugging method had beneficial effects and the students that read the document were able to correct a significantly larger proportion of the bugs.

### **2.3.3 Other approaches**

Several other studies tried to solve these issues with similar approaches. Böttcher et al. (2016) taught debugging skills to students by using a text and a demo which involved an explanation of the debugger and a debugging method that used the Wolf Fence algorithm (Gauss, 1982). This is a simple algorithm that finds the bug by using a binary search; put a breakpoint in the middle section of your code and see in which part the bug occurs. Repeat this process recursively in the part where it goes wrong until the bug is found and fixed.

## **2.4 Effectively teaching new material: direct instruction**

The aforementioned studies taught the debugging teaching material through an presentation or demo (Michaeli & Romeike, 2019), a written document (Allwood & Björhag, 1991) or a combination of these (Böttcher et al., 2016). All of these had a positive impact on the students, but other studies focused on how to effectively teach study material in general, which are interesting to mention as well.

One particular principle that is seen as effective and educationally enhancing is direct instruction. Direct instruction is a highly interactive approach to teaching as opposed to most modern approaches. The lessons are structured in such a way that the teacher is able to see whether students understand the explained concepts directly after the explanation. If this is not the case, the teacher is able to guide the students to a better comprehension immediately. As a result, the students are actively involved with the

lesson and they will understand the material better and quicker (Gersten & Keating, 1987).

### **2.4.1 Applying direct instruction**

To accomplish the characteristics of direct instruction, the following steps should be integrated into the educational occurrence (Renard, 2019):

1. Introduction/review  
This is the opening of the lesson and is intended to engage the students. This could be done by referring to the previous lesson or providing some lesson objectives.
2. Present the new material  
To effectively present the new material, the teacher should use clear and guided instructions and the lessons should be carefully organised step-by-step. The presentation could be done through a lecture or a demonstration.
3. Guided practice  
In this phase, the teacher and students practice the material together, to guide initial practice, reteach if necessary and correct common mistakes.
4. Feedback and correctives  
The teacher gives feedback which depends on the previous phase, e.g. if a questions were correctly answered, quickly move on to keep the pace of the lesson, but if the questions were incorrectly answered, provide some hints.
5. Independent practice  
In this phase, students apply the new learning material on their own. This practice provides the repetition necessary to understand the subject and also helps the students to become automatic in the use of these new skills.
6. Evaluation/review  
In the last phase, an evaluation method is used to check whether the students actually understand the newly acquired knowledge and can apply it correctly. If this is the case, students learned the new material and the teacher can move on to the next subject.

Beside these steps, several other features also enhances direct instruction (Binder & Watkins, 1990):

- Use small groups  
The ideal group size is around 5 to 10 students, because this size provides the best one-to-one instruction and feedback.

- Unison responding  
To generate response from all the students, it is useful to use a tool such that all the students are able to respond at the same time. This provides the students with the greatest amount of practice and the teacher has the most information about each individual student.
- Keep the pace high  
Rapid pacing is very useful, because it allows the teacher to present more material and spent more time on the practice sessions. In addition, a rapid pace helps the students to keep their attention to the current task, which improves the learning process.

## 2.4.2 Benefits of direct instruction

There are a lot of positive effects associated with the application of direct instruction. In a study where they compared all the different teaching programs, direct instruction was the most effective on basic skills achievement, self-concept and cognitive skills (Watkins, 1988). Another study concluded that using direct instruction in class lead to better achievements and a lower drop-out rate across four different communities (Gersten & Keating, 1987). Moreover, other positive effects of direct instruction were significantly higher scores from students than students that did not got taught with this principle (Binder & Watkins, 1990).

In sum, direct instruction is a great principle with many benefits and should therefore be used in an educational intervention if possible.

## 2.5 Measuring the effect of an educational intervention

Different methods are used to test the effects of the educational intervention about debugging, but exercises, observations and interviews/questionnaires are the most used among these studies (or a combination of these). The exercises consist of debugging exercises; programs which contains a number of different errors and the students are asked to find and fix these errors. The answers of the students can be used directly for an analysis, e.g. comparing the errors which were of weren't solved of the test and control group (Allwood & Björhag, 1991; Michaeli & Romeike, 2019). Moreover, Böttcher et al. (2016) asked the students to explicitly write down their process and provide screenshots of their used breakpoints, which could be used for a more extensive analysis.

Katz and Anderson (1989) observed their subjects when they were making the debugging exercises and were focusing on the most occurring problems and how the students solved these.

Michaeli and Romeike (2019) used debugging exercises to test the impact of the intervention on the performance of the students, but they used questionnaires to get a better insight in the debugging self-efficacy of the subjects. In their previous study, Michaeli and Romeike interviewed several teachers to identify the underlying problems regarding debugging in the classroom.



## Chapter 3

# Goal

### Research Question

The main reason that debugging is underrepresented in education is that the teachers don't have the time and lack the teaching material to explicitly teach debugging. To tackle this problem, we identified how to effectively teach debugging within a programming course. Teachers can integrate these elements into their lecture, to make the lecture effective and efficient. This lead to the following research question:

*How can we effectively teach debugging within a programming course?*

### Research Subquestions

This project has the outline of a design research (van den Akker, McKenney, Nieveen, & Gravemeijer, 2006), in which we design a a learning activity and performed it in a lecture of a programming course at the Radboud University in Nijmegen. Afterwards, we evaluate the learning activity and with these things combined, this research results in design principles for the education about debugging and knowledge about how the students learn. The corresponding research subquestions are:

1. *What are elements of an effective learning activity about debugging in a programming course?*
2. *How can the students' resulting debugging skills be characterised?*

## Chapter 4

# Methodology: Design

We designed an online learning activity to teach a systematic debugging procedure and useful supporting strategies for debugging to students. These strategies were taught in a lecture that was based on the principles of direct instruction, a teaching strategy which is proven as very effective.

The following steps were taken to design our learning activity and to achieve the aforementioned goals. Some criteria is given to see if the design actually meets our given requirements.

### 1. Setting up the learning goals

We taught the students a systematic debugging procedure, such that the students are able to debug their programs in a systematic manner. Additionally, several debugging strategies were presented in this lecture that support this debugging procedure. To see which strategies were the most useful for the students, we looked at the papers written by Altadmri and Brown (2015) and Böttcher et al. (2016). These authors looked at the most frequently encountered programming problems by students and we chose the debugging strategies that solve the most occurring problems.

### 2. Designing a lecture based on these learning goals

The debugging procedure and the supporting strategies were taught in an online lecture with presented slides. As decided, this lecture was built around the principle of direct instruction and to achieve this, the mentioned steps (section 2.4) should be integrated in the lecture and several exercises and programs had to be designed to achieve this. The presentation started with an introduction which gave an overview of the presented material and enthused the students for the lecture. The taught material was tested after the explanation to see if the students understood the material and the students were able to ask questions if necessary. To achieve this, we gave a demo after the explanation and after most strategies, the students had to make a small exercise to see if

they understood the material. At the end, there was an opportunity for independent practice. Therefore, we designed an debugging exercise with which the students could practice.

With these concepts, direct instruction was integrated into our presentation and so, the material was taught effectively to the students and the other benefits of direct instruction applied to the designed lecture.

### **3. Designing demos for the strategies**

The designed demos were programs that contained a bug that was relevant to the explained concept. Therefore, the presented strategy was applied in practice in a program inside the IDE, which was a familiar environment for the students. These demos served as an example and worked better than code snippets presented on slides.

### **4. Designing intermediate exercises for immediate practice**

The small exercises after the demos consisted of a multiple choice question with a picture which was related to the corresponding strategy. The students had to answer this question to quickly practice with the material.

### **5. Designing an exercise for independent practice**

We provided a debugging exercise which involved different types of bugs with which the students could practice after the lecture. The bugs in this exercise should involve all the presented debugging strategies, such that the students could practice with all the learning goals.

### **6. Determining which tools to use for the learning activity**

We needed an online environment in which the presentation could be given and it was important that the designed slides could be seen by the students. It was also convenient if the lecture could be recorded, such that the students could see the recording afterwards.

The questions used for the intermediate exercises, were given through an online quiz environment, for which we had a few requirements. It was important that all the students answered individually and all at the same time, which is also referred to as Unison responding (chapter 2.4). Moreover, it was also important that the teachers saw the results of the students in real time, to give some more explanation if necessary. Lastly, it was also convenient that we had the option to export the results to an excel file afterwards, to analyse the results more easily.

After following these steps, we created a learning activity that covers a systematic debugging procedure and covers the most relevant supporting debugging strategies. These taught concepts were given in an online presentation which was based on direct instruction that included demos and intermediate questions through an online quiz environment. Next to this

presentation, a debugging exercise was available to the students to practice with the presented concepts.

## Chapter 5

# Results: Design

The lecture was designed by following the steps described in chapter 4:

### 1. Setting up the learning goals

The debugging procedure described by Michaeli and Romeike (2019) was used in the presentation as the systematic debugging procedure covered in the lecture, because it is relatively simple and very effective. The concrete supporting debugging strategies presented in the lecture were based on the most frequently occurring problems for students, which are the following:

- (a) Identifying correct types for arguments or return objects (Altadmri & Brown, 2015)
- (b) Evaluating a run time error (Böttcher et al., 2016)
- (c) Finding the relevant lines of code that cause the bug (Altadmri & Brown, 2015), (Böttcher et al., 2016)

The following debugging strategies were chosen to be presented:

- JavaDoc
- Systematically evaluating a run time error
- The debugger (basic functionality and the different commands)

These debugging strategies were chosen, because these tackle the most frequently occurring problems. The documentation in JavaDoc will solve the first problem, because this provides the necessary information to determine the needed types. JavaDoc also provides more information about functions in general, which can be used to solve other bugs as well. A systematic evaluation of a run time error will solve the second problem. This evaluation consists of fixing the bug by determining the relevant lines and type of error by inspecting the given error. The debugger could support the students with the latter 2 problems, but is a useful tool in general. Students are often afraid to use

the debugger, but we give a simple explanation which should solve this.

After deciding which strategies were taught, the learning goals of the lecture could be specified as these:

- Students apply the systematic debugging procedure of Michaeli and Romeike when debugging
- Students know about the JavaDoc inside NetBeans
- Students systematically evaluate a run time error to determine the relevant lines in the code and solve the bug
- Students understand the basic functionality of the debugger

## 2. Designing a lecture based on these learning goals

The different phases of direct instruction were integrated into the lecture:

- The introduction consisted of an overview of the content and an online quiz to enthuse the students and let them be familiar with the online quiz environment.
- All the material was presented with the designed slides and students could ask questions if necessary.
- For all the taught strategies, a demo was made to see a concrete example in the programming environment. This was a combination of presenting the new material and guided practice, because it was interactive. The demos are discussed in more detail in the next step.
- For most strategies, there was a question for the students to test if they understood the material. This question served as more guided practice for the students and based on the results, we could give feedback and apply correctives if necessary. The questions are discussed in more detail in step 4.
- At the end of the session, the students could practice with a debugging exercise such that there was an opportunity for independent practice afterwards. The debugging exercise is discussed in more detail in step 5.

The last step, evaluation/review, which should happen after independent practice, was not integrated into the lecture, because it was not possible to speak to all the students after the independent practice. The used slides in the presentation can be found in appendix C.7.

A recording of the presentation was made available for the students as well as the used slides, such that the students could refer back to the presented material if necessary.

### 3. Designing demos for the strategies

The program used for the demo about JavaDoc can be found in appendix B.3. It consisted of a bug which could be fixed by looking at the JavaDoc of the function trim().

The program used for the demo about evaluating a run time error can be found in appendix B.4. It consisted of a run time error caused by a null pointer exception, which could be fixed by systematically evaluating the given error.

There were 2 demos for the explanation about the debugger. The first demo can be found in appendix B.5. There was a simple bug in this program, but the main purpose was to make the students familiar with the basic functionality of the debugger and how convenient it was to see all the variables. The second demo can be found in appendix B.6. There was no bug in this program, but the different lines were structured in such a way that the difference and purpose of the different commands became clear for the students.

### 4. Designing intermediate exercises for immediate practice

Question 6 in the quiz (handout can be found in appendix C.4, the used picture can be found in figure A.3) served as the small exercise for JavaDoc. The JavaDoc of a user defined function was shown that caused a bug and students had to indicate how to probably solve this bug. This tested if students could use the provided JavaDoc to solve the bugs.

Question 7 in the quiz (the used picture can be found in figure A.4) served as the small exercise for evaluating a run time error. A run time error was shown and students had to indicate what was probably the error and on which line the bug was probably located. This tested if the students could identify the run time error type and the location with only the provided error.

No question was made for the debugger, because it was inconvenient and the independent practice served as a really good exercise for the debugger.

### 5. Designing an exercise for independent practice

The debugging exercise can be found in appendix B.1 and the corresponding answers can be found in appendix C.1. It consisted of 4 compile time errors, 2 run time errors and 4 logic errors, so all the different types of errors were present in the exercise to practice with the presented debugging process. The different bugs could be solved with the presented debugging strategies, so the students could also practice with those and therefore, all the learning goals were addressed in this exercise.

## 6. Determining which tools to use for the learning activity

The used online environment was Zoom. The slides could be presented in Zoom, such that these were clear for the students and the presentation could be automatically recorded, which was really convenient.

The online quiz environment Socrative was used for the quiz. All the students could easily join and could answer the questions individually and at the same time. The teacher could also see the results in real time to apply correctives if necessary and the teacher was also able to export the results to an excel file. Therefore, Socrative served as a great platform to use and the handout of the quiz can be found in appendix C.4

The complete lecture is represented in table 5.1, including the corresponding direct instruction phases, demos and questions.

Table 5.1: The designed lecture

	Activity	Phase	Used programs/tools
1.1	Overview of the content	Introduction	-
1.2	Quiz	Introduction	Socrative, questions 1-5
2	Debugging procedure	Present the new material	-
3.1	JavaDoc: Explanation	Present the new material	-
3.2	JavaDoc: Demo	Present the new material/Guided practice	JavaDocDemo
3.3	JavaDoc: Quiz	Guided practice/Feedback and correctives	Socrative, question 6
4.1	Tips evaluating run time error: Explanation	Present the new material	-
4.2	Tips evaluating run time error: Demo	Present the new material/Guided practice	RunTimeDemo
4.3	Tips evaluating run time error: Quiz	Guided practice/Feedback and correctives	Socrative, question 7
5.1	Debugger: Explanation basics	Present the new material	-
5.2	Debugger: Demo	Present the new material/Guided practice	DebuggerDemo1
5.3	Debugger: Explanation different commands	Present the new material	-
5.4	Debugger: Demo	Present the new material/Guided practice	DebuggerDemo2
6	Overview of the content (repetition)	Feedback and correctives	-
7	Debugging exercise	Independent practice	DebuggingExercise1



## Chapter 6

# Methodology: Evaluation

After designing the learning activity, we evaluated the effects of it to see if it was effective. This evaluation consisted of 2 parts: evaluating whether the learning goals were achieved and evaluating what were the pros and the potential improvements for the designed lecture. Therefore, the design of these evaluations were based on the aforementioned learning goals, which were the following:

- Students apply the systematic debugging procedure of Michaeli and Romeike when debugging
- Students know about the JavaDoc inside NetBeans
- Students systematically evaluate a run time error to determine the relevant lines in the code and solve the bug
- Students understand the basic functionality of the debugger

We had three moments where we collected response from the students: a pre-test at the presentation, several interviews a week after the presentation and second interviews 3 weeks after the presentation. The pre-test was integrated into the introduction of the presentation and so, we collected data from all the watching students. We interviewed 5 students in the first interview to see if they applied the learning goals in a debugging exercise and they were asked about the pros and improvements for the lecture. These students were all currently following the course Object Oriented Programming. However, for several learning goals it was sufficient if the students knew about the strategy (JavaDoc and the debugger), but the first interview only observes whether the students applied the taught debugging strategies, which is not the corresponding learning goal. Therefore, these 5 students were also contacted for the second interview to see if they applied several learning goals in practice with the programming exercises and if they know how to apply them, to evaluate the learning goals regarding JavaDoc and the debugger in more detail. The result of these evaluations were combined and

analysed afterwards to investigate the pros of this presentation, the effect of the intervention and how this presentation could be improved.

## **6.1 Pre-test**

### **6.1.1 Data collection**

At the start of the educational intervention, a small quiz was held in Socratic to give an indication of the current debugging skills of the students. A printout of this quiz can be found in appendix C.4. Several strategies were presented and the participants had to indicate whether they used these strategies when they are debugging with the options "yes", "no", and "no opinion". These strategies corresponded to the explained concepts in the educational intervention. The usage of the following strategies were asked in the quiz:

- Searching on Stack Overflow
- Looking at documentation online
- Looking at documentation inside the IDE (NetBeans)
- Print statements to see the values of the variables
- The debugger

### **6.1.2 Data analysis**

Afterwards, the answers of all the students to this quiz could be exported to an excel file which was analysed. We removed the students that did not answer all the questions, such that all the students that were left participated in all the questions. We counted the answers the students gave to a specific question to see whether the majority of the students already used that debugging strategy. This gave an indication which strategies the students already used and which strategies they were not using. If a debugging strategy is used by only a minority of the students, but several students did use that debugging strategy in one of the interviews, it is safe to say that the students learned that debugging strategy in the lecture, which helps to identify how the lecture affected the debugging skills of the students.

## **6.2 First interviews**

### **6.2.1 Data collection**

These online semi-structured interviews were held on a platform chosen by the students (Skype, Zoom, Discord etc.). These interviews mostly consisted

of students solving a debugging exercise and answering a few questions while they were observed through a screen share. In particular, we observed the strategies that the participants applied to fix the individual bug, the order in which the bugs were fixed and the overarching debugging procedure. Afterwards, we looked how the applied strategies corresponded to the learning goals to see the effects the learning activity had on the students.

The debugging exercise was shared at the start of the interview with the interviewees through a zip folder. The interviewees could import the project to NetBeans and were asked to fix all the bugs on their computer and therefore, the students had access to all the tools they normally used in practice for debugging and they were comfortable with the used environment.

The structure of the interview was as follows:

1. Question: what were the pros of the presentation?
2. Question: how could the presentation be improved?
3. A debugging exercise with 5 bugs

### **Pros and improvements**

The purpose of the first 2 questions was to see what the participants thought about the designed lecture. These answers gave us an insight in what the students thought about the lecture and thus, which elements of the presentation could be used in other lectures and which aspects could be improved. These questions also served as a warming-up and an ice-breaker.

### **Achievement of the learning goals**

Afterwards, the students had to solve a debugging exercise. The students were observed to see which strategies and tips they applied from the learning activity to evaluate whether the learning goals were achieved. The specific debugging exercise used in this interview can be found in appendix B.2 and the corresponding solutions in appendix C.2.

The program took an array of integers and calculated the median of it. This was done by a function `calcMedian`, that took the middle elements of a sorted list, the function `sortNumbers` that sorted a list of numbers using selection sort supported by the last function, `findIndexWithSmallestNumber`, which returns the index with the smallest number. We also provided 4 test cases which the interviewees could use to see if they correctly fixed a bug the program. There were 5 bugs in the program, which were all quite unique and all the different type of bugs occurred (1 compile time error, 1 run time error and 3 logic errors). We observed the order the bugs were fixed, what the overarching debugging procedure was that the participant applied, the fixes for the individual bugs and other interesting, maybe unforeseen observations. Every different type of bug had a different purpose

regarding the learning goals, so these will be discussed in more detail in the next paragraphs, but first is discussed why the order in which the bugs were solved and the overarching debugging procedure were observed as well.

#### **Order in which the bugs were solved & overarching debugging procedure - The presented systematic debugging procedure**

With these 2 observations, we could determine whether the learning goal regarding the debugging procedure of Michaeli and Romeike (figure A.1) was achieved. This 2 characteristics of this procedure are the order of the 3 different phases (first Compile, Run afterwards and Logic at last) and the cycles within these phases (roughly speaking, executing the program, determining what went wrong, finding the relevant lines of code, fixing it and executing the program again and repeat if necessary). With the order in which the bugs were solved, we could determine whether the students firstly solved the compile time error, the run time error afterwards and the logic errors at last.

Within the overarching debugging procedure, we noted when a participant executed a program and which test cases failed, what their next actions were and which functions they inspected. We determined from these actions whether the participant applied the described cycles in the debugging procedure by observing if the participants executed the test cases, determining what went wrong and finding the relevant lines of code, fixing the bug afterwards and repeating this cycle until all the bugs were fixed. The cycles for the different phases are slightly different, but the cycles could roughly described as this. So, with these 2 mentioned observations we could determine if the participant applied the systematic debugging procedure presented to test the first learning goal.

#### **Compile time error - JavaDoc**

The first bug was compile time error bug that consisted of an incorrect type that was located in calcMedian in line 29. The called function sortNumbers returns an array and is stored in the variable sortedNumbers, which has the type ArrayList<Integer>. These types cannot be automatically cast and therefore, the following error message is provided by the IDE and shown when the program is executed: incompatible types: int[] cannot be converted to ArrayList<Integer>. The most practical solution would be changing the type of sortedNumbers to int[] and could be found by inspecting the types of the functions, for which JavaDoc is the perfect tool. Therefore, this bug served as a good indication whether the students were familiar with JavaDoc to test the second learning goal.

#### **Run time error - Systematically evaluating a run time error**

The second bug was a run time error that was caused by a loop in sortNumbers in line 44. This loop iterated over the array named list to sort the array, but started on index 0 and stopped when the index was greater than list.length. However, list.length is an invalid index and so, the run time error is thrown with the corresponding line and an indexOutOfBoundsException-

ception when the program executed. This most practical fix for this bug would be to change the `<=` to a `<` such that the index `list.length` is not inspected anymore. We observed how the student evaluated the run time error by looking at their actions (determining the relevant lines of code and determining what the probable cause is) and compared this to the presented systematic evaluation to see whether the third learning goal was achieved.

#### **Logic errors - The debugger**

The last 3 bugs were logic errors; the third bug was caused by a mistake in the if condition in line 30 in the function `calcMedian`, where is determined whether the list has an even or odd length to see if 1 or 2 elements should be returned. This is done by checking if the `length mod 2` is 1 or 0, but the case is exactly switched around, such that an even length returns 1 element instead of 2. This bug could be solved by changing the 1 to a 0 or switching the code blocks within the if and the else.

The fourth bug was caused by an off-by-one error in the function `calcMedian` in line 30, in the previously mentioned if case where 2 middle elements are returned of an array with an even length. Currently, the elements at the indices `(numbers.length/2+1)` and `(numbers.length/2)` are returned, but these are the incorrect indices (e.g. a length of 6 returns in this case elements at indices 4 and 3, but should return elements at indices 2 and 3). This bug is solved by changing these indices to the correct indices: `(numbers.length/2-1)` and `(numbers.length/2)`.

The last bug was caused in the function `findIndexWithSmallestNumber` in line 63. In this function, the program iterates over the given array and stores and returns the index with the smallest element by comparing the inspected number with the currently smallest number. However, in the if condition where this comparison is made, the current result is replaced with the inspected number if the inspected number is bigger than the result, which means that not the index with the smallest number is returned, but the index with the biggest number. As a consequence, the function `sortNumbers` sorts the array descending instead of ascending. This has no consequence on the median of arrays with an odd length, but the elements of the median of an array with an even length are switched around, which results in incorrect test cases. This most practical fix for this bug is to change the `>` to a `<` or a `<=`.

Some of these bugs became pretty clear from the different test cases, e.g. bug 3, but other bugs were more difficult to find in the code, e.g. bug 5. So, these 3 logic errors served as a perfect opportunity for the participants to apply the debugger, because they could easily inspect the sorted list and check whether the functions functioned correctly by walking through the function step-by-step. Therefore, we could test the fourth learning goal by observing whether they used the debugger at these bugs and so, all the learning goals were evaluated in the debugging exercise.

## Structurally collecting data

To structurally collect data from these interviews, a schema was used to note all the aforementioned observations, namely how all the individual bugs were solved in the debugging exercise and which strategies were applied by the participants, the order in which the bugs were solved, the overarching debugging procedure, the answers to the 2 questions and other interesting notes. The schema used for this can be found in figure C.3.

As a result, the interviews resulted in 5 filled in schemes of the interviewees that represented how they tackled a debugging exercise and what they thought about the given lecture. These observations gave us insight if we achieved the learning goals, what the pros were of the lecture and how the students thought it could be improved. This collected data could directly be used for the analysis in Atlas.

### 6.2.2 Data analysis

To analyse the collected data, we used the tool Atlas to create codes which we can use to label different observations. To determine the different codes, we used a combination of a bottom-up and top-down approach. We already had an idea about observations that would probably occur, namely the learning goals (e.g. using the debugger, using the systematic evaluation of a run time error), but we could also identify other characteristics which we did not foresee or expect, which would result in other codes. With these codes, we could reason more easily about the observations. A more detailed description of our analysis is described in the following paragraphs.

#### Pros and improvements

We coded all the mentioned pros of the students to see the frequently mentioned pros. We already had some idea of codes that could occur as we thought that they would probably occur, because they were quite unique for this lecture (e.g. demo was useful, interaction was useful), but other codes had to be derived of the answers.

The mentioned improvements could be coded, but these are probably too specific to be grouped together. Therefore, we thought it was better to inspect every individual improvement and see if this could be integrated into a possible improved iteration of the lecture. However, if it was possible to code the improvements, the codes would be determined in a bottom-up approach, because we had no idea which improvements the students would be mentioning.

Afterwards, the result of this analysis would be several pros that were frequently mentioned by participants and all the improvements that were indicated by the students.

## **Achievement of the learning goals**

### **Order in which the bugs were solved & overarching debugging procedure - The presented systematic debugging procedure**

The order in which the bugs were solved did we code as the order of the presented debugging procedure if the order was 1-2-(3-4-5), where 3, 4 and 5 could be in any order. In this order, the Compile phase is applied firstly, the Run phase afterwards and the Logic phase at last, which is indeed the presented order. If the participant did not have this order, the order was classified as not the order of the presented debugging procedure.

Within the overarching debugging procedure, we marked the different debugging phases if we recognised the corresponding cycles in the debugging procedure. For example, if the participant executed the program, evaluated the test cases and tried to fix the bug and executed it again until all the logic errors were fixed, this phase is marked as the Logic phase. The same was done for the Compile and Run phase.

If a participant applied the correct order and applied all the different debugging phases from the lecture, we could state that that participant applied the presented debugging procedure. If a majority of the participants (4 out of 5) applied the presented debugging procedure, we could state that the corresponding learning goal was achieved.

### **Compile time error - JavaDoc**

We coded the debugging strategies that the participants used to fix the compile time error (bug 1). Some codes that would probably occur were "Using JavaDoc" and "Fixed compile time error by hand", but other codes could also be identified, which we hadn't expected. If at least 4 out of 5 participants used JavaDoc as a debugging strategy to solve this bug, we could state that most of the participants applied JavaDoc while debugging. However, the learning goal is not about applying JavaDoc, but knowing how to use it, because it is not always necessary to use JavaDoc. If indeed the majority of the participants used JavaDoc, the corresponding learning goal is also achieved, but if this is not the case, it doesn't necessary mean that the corresponding learning goal is not achieved. Therefore, the achievement of this learning goal will be evaluated in more detail in the second interview, where we specifically ask if participants know how to use JavaDoc.

### **Run time error - Systematically evaluating a run time error**

The presented systematic evaluation of a run time error consists roughly speaking of 3 steps: Determining the relevant lines from the message, determining what is probably the cause from the given exception and fixing the bugs with this information. These steps can be recognised in the debugging strategy applied by the participants to solve the run time error (bug 2). If the participant used the run time error to identify the location, than that is coded as "Found incorrect line by inspecting run time error". If the participant determined the type of exception and went to the relevant line, it

should be able to identify the bug rather quickly and solve it immediately, because it is a relatively common bug. If this was observed, this is coded as "Immediately saw the fix for the bug". If both of these codes applied to a participant, that participant applied the systematic evaluation of a run time error and if at least 4 out of 5 students applied this evaluation, the learning goal regarding the evaluation of a run time error was achieved.

#### **Logic errors - The debugger**

The logic errors (bugs 3, 4 and 5) were grouped to evaluate whether the students apply the debugger when debugging. It really depends on the participant when it needs the debugger to solve a bug or if the bug is easy enough to solve without the debugger. That was the reason why there are 3 different logic errors in the exercise, such that it is probable that there is at least one bug for which the participant would need the debugger. Similar as before, we coded all the different strategies used by the participants, which were codes we already expected (e.g. using the debugger, go through the code line-by-line in your head, immediately fixing the error), but other codes could also be created, which we didn't expect beforehand.

If a participant used the debugger to solve at least 1 bug, we stated that that participant applied the debugger in this debugging exercise, because of the previously mentioned variety regarding the debugging skills of the participants. If at least 4 out of 5 participants applied the debugger in this debugging exercise, the corresponding learning goal was achieved. However, similar to JavaDoc, the learning goal is that students know how to use the debugger and not applying it, because it depends on the student if it is actually necessary to apply the debugger. So, even if only the minority of the students use the debugger, it does not mean that the learning goal isn't achieved, but this learning goal will therefore be evaluated in more detail in the second interview, where the students are explicitly asked if they know how to use the debugger.

## **6.3 Second interviews**

### **6.3.1 Data collection**

To see whether several explained principles were also used in practice by the participants, we contacted the same participants that were interviewed a few weeks later to ask which concepts they applied in practice. The following questions were asked:

- Did you use JavaDoc within the exercises last weeks?
- If no, would you know how to use JavaDoc?
- Did you use the debugger within the exercises last weeks?
- If no, would you know how to use the debugger?



We did not ask about the systematic debugging procedure and the systematic run time error evaluation, because it was pretty clear from the results of the previous interviews that these learning goals were already achieved. The answers to these questions were mostly yes or no with a bit of elaboration and therefore, it was possible to create a table with the participants, the questions and their yes or no answer, which could be analysed. We assumed that whenever a participant used a strategy, it also knew how to use it, even though this was not explicitly mentioned by the participant.

### **6.3.2 Data analysis**

From this table and, if necessary, the several elaborations, we could evaluate the learning goals related to JavaDoc and the debugger in more detail. The number of participants that used JavaDoc gave an indication whether students used JavaDoc in practice, but the number of participants that would know how to use JavaDoc is more important to the learning goal. If at least 4 out of 5 participants would know how to use JavaDoc, we achieved the learning goal regarding JavaDoc, even though the minority of the participants actually used JavaDoc in the first interview. The similar holds for the debugger; the number of participants that used the debugger indicated whether students used the debugger in practice. If at least 4 out of 5 participants would know how to use the debugger, we achieved the fourth learning goal about the debugger.

## **6.4 Combining the results**

The results of the different evaluations were combined to answer the research (sub)questions. The pros mentioned by the students in the first interviews were used to determine how the students thought of the particular elements of the designed learning activity for the first research subquestion. The pre-test, the observations regarding the debugging exercise from the first interview and the results of the second interview were combined to answer the second research subquestion. The pre-test gave an indication which debugger strategies the students already used before the lecture, the first interviews indicated which debugging strategies the students used after the lecture and the second interviews indicated which strategies the students used in practice and if they know how to use it. With these results it was possible to evaluate all the learning goals and see how the lecture affected the debugging skills of the students. The improvements mentioned by the students in the first interviews were used in the discussion to give some possible improvements for a next iteration of the design.

## Chapter 7

# Results: Evaluation

### 7.1 Pre-test

58 participants participated in the pre-test at the start of the lecture and the result can be found in table 7.1.

Table 7.1: The result of the pre-test

Do you use the following strategy when debugging?	Yes	No	No opinion
Searching on Stack Overflow	52	5	1
Looking at documentation online	52	5	1
Looking at documentation inside the IDE (NetBeans)	15	38	5
Print statements to see the values of the variables	53	4	1
The debugger	15	40	3

As can be seen, the majority of the participants use Stack Overflow and read documentation online, but most of them do not use the documentation provided in NetBeans, which is JavaDoc. Furthermore, most participants use print statements to find bugs, but a majority of them does not use the debugger. So, if we notice that the debugger and JavaDoc are used at the interviews and the interviewees actually used these strategies after the presentation in practice, the lecture had its desired effect, namely that the students applied the taught strategies.

### 7.2 First interviews

The results of the interviews were analysed and the learning goals and other interesting observations are discussed in this section. The filled in interview schemes can be found in appendix C.5.

### 7.2.1 Achievement of the learning goals

### 7.2.2 Pros and improvements

Several pros about the lecture were mentioned by the participants, but one was particularly common in the interviews: the interaction within the lecture. Not only kept the interaction the participants engaged with the presentation (mentioned by participant 4), but most participants also said that the interaction was really useful. This interaction was achieved by integrating direct instruction into the lecture, especially the questions within the quiz and the demos, which were both indicated as very useful by most interviewees. Participants 1,3 and 5 also mentioned that the explanations were clear, which was probably the result of this integration. In short, the participants were very positive about the elements of direct instruction and therefore, we advice teachers to integrate direct instruction into their lectures if possible by having small exercises between different material for example. Moreover, programming teachers could have more demos within their lectures, because students appreciate these more than code snippets in slides.

Next to these positive points, students only had a few minor improvements for the lecture. Participant 1 indicated that some pictures were rather small in the slides. Participant 3 mentioned that it would be appreciated if there would be more options within the quiz (in particular the first 5 questions, a "sometimes" options would have been nice) and participant 2 said that it would be useful if the students could see the intermediate results of the quiz questions. This was actually possible in the quiz environment, but we simply forgot to show it, but we keep this in mind for the next time. In sum, interviewees were really positive about the presentation and had a few minor improvements.

### The presented systematic debugging procedure

To evaluate if this the learning goal "*Students apply the systematic debugging procedure of Michaeli and Romeike when debugging*" was achieved, we looked at the order in which the bugs were fixed and if the students used the presented cycles within these phases. The order in which the all participants solved the bugs had a clear pattern; the compile time errors firstly, the run time errors afterwards and the logic errors at last. This was also the recommended order of Michaeli and Romeike, so the interviewees applied the presented order of debugging phases.

Moreover, it was also clear from the overall debugging procedure of the participants that they applied the presented debugging cycles. As can be seen in the schemes, the Compile, Run and Logic cycle were applied, but the latter cycle is the most interesting to be discussed in more detail. After all the compile and run time errors were solved, all the participants executed

the program and compared the results to the desired results. They said what was probably wrong and started to investigate the relevant lines of code or look for them with the aid of the debugger for example. After they fixed (or tried to fix) the bug, they executed the program again to see if the bug was solved or not and repeated the cycle until all the bugs were fixed, which is exactly the presented logic cycle. Therefore, all the participants applied the presented debugging procedure and thus, the first learning goal was achieved.

### **JavaDoc**

To evaluate if the learning goal "*Students know about the JavaDoc inside NetBeans*" was achieved, we looked at the strategies used by the participants to solve the first bug; the compile time error. Only participant 2 used JavaDoc to get more information about the function causing this bug and afterwards solved it by applying the provided quick fix by the IDE. Participant 1 immediately solved the bug by using the provided quick fix and the other 3 participants said it was an incorrect type from the provided error message and fixed it by hand. From this can be included that the second learning goals is not completely achieved, because only 1 participant applied this debugging strategy and therefore indicated that he knew about JavaDoc. However, the clear error messages and quick fixes provided by the IDE were probably the reason that many participants did not use the JavaDoc within the debugging exercise, because using JavaDoc was a bit redundant. So, from this first interview it is not completely clear whether this learning goal was achieved, but the second interview gave more insight into this.

### **Systematically evaluating a run time error**

To evaluate if the learning goal "*Students systematically evaluate a run time error to determine the relevant lines in the code and solve the bug*" was achieved, we looked at the strategies used by the participants to solve the second bug; the run time error. After solving the compile time error, all the participants executed the program and got the run time error. They read it out loud and determined the corresponding line which was provided by the run time error. They also mentioned that it probably had to do with indices, because of the provided `indexOutOfBoundsException`. Afterwards, all the participants immediately fixed the bug, because it was a common mistake made by programmers, as several interviewees indicated as well. These observations indicate that all the participants applied the presented evaluation of a run time error and thus, the third learning goal was achieved. Moreover, participant 5 mentioned in particular that the provided list with the different types of run time errors and the corresponding fixes was really

useful, so this information was also appreciated.

### **The debugger**

To evaluate if the learning goal ”*Students understand the basic functionality of the debugger*” was achieved, we inspected how the different participants approached the remaining 3 bugs; the logic errors.

After solving the run time error and stating that the test cases were incorrect, participant 1 immediately used the debugger to inspect the whole program step-by-step and corrected the program in a bottom-up manner by firstly solving the independent function and the afterwards the functions that depend on the now correct function and so on. With the assumption that the called functions are correct, the interviewee could fix all the functions one by one until all the functions were correct and as a result, all the bugs were fixed as well. Participant 1 therefore used the debugger to solve all the logic errors and thus knows how to use the debugger.

Participant 4 was the only participant that didn’t use the debugger for these bugs, but one print statement to see whether the sorting function was correct. This participant also mentioned that (s)he was more familiar with print statement and already used the debugger when necessary, even before the lecture. Additionally, this participant mentioned that (s)he was a relatively good programmer and solved all logic errors in the exercise almost immediately after determining what was probably wrong and so, it is safe to say that (s)he didn’t use the debugger, because the exercise was relatively easy for this participant. However, this participant did mention that s(he) understood the basic functionality of the debugger.

The other 3 participants solved at least one bug of these bugs with the aid of the debugger and therefore understood the basic functionality of the debugger. The other bugs were either immediately solved, because the participants mentioned that it was an obvious or common bug, or the participant mentioned that they went step-by-step through the code in their heads and solved them by noticing the incorrect logic. Therefore, these participants understood the workings of the debugger, but only used them when necessary.

Moreover, almost all the interviewees (except for participant 4) indicated that they never used the debugger before and will probably use them when debugging in the future after the explanation in the designed lecture.

So, it is safe to say that the debugger was the most useful subject of the lecture for students and that the fourth learning goal was achieved as well.

### **7.2.3 Another interesting observation**

Next to the learning goals, pros and improvements, there was one other interesting observation, namely how the students used the different test

cases to determine the bugs. Participant 1 solved the test cases one-by-one by inspecting the whole program with the debugger, as mentioned before. However, the other 4 participants partly determined the location of the bug by inspecting the incorrect test cases and comparing them to identify the underlying problem. After they had no clue anymore, they started inspecting the functions and determined which were not correctly functioning, thus applying the similar strategy as participant 1.

The interesting observation are the different approaches used by the participants to determine the bug given the failed and succeeded test cases. The designed lecture mostly focuses on finding or fixing bugs, but interestingly, the lecture could also be expanded with some tips about using the actual test cases to find the locations of bugs. Moreover, the lecture could be even more extended by teaching the students how to write effective and efficient test cases, which as a result can be used to find some bugs even faster. As a result, this will result in a lecture where debugging and testing are covered, which are two very important assets for a programmer, but could be given more attention in education.

### 7.3 Second interviews

The questions asked in the second interview were answered by the participants over chat and their responds can be found in appendix C.6. The mentioned participants correspond with the same participants from the first interview. These results are presented in table 7.3.

Table 7.2: The result of the second interview

	P1	P2	P3	P4	P5
Q1: Did you use JavaDoc within the exercises last weeks?	No	No	Yes	Yes	Yes
Q2: If no, would you know how to use JavaDoc?	Yes	Yes	Yes	Yes	Yes
Q3: Did you use the debugger within the exercises last weeks?	Yes	Yes	No	No	No
Q4: If no ,would you know how to use the debugger?	Yes	Yes	Yes	Yes	Yes

3 of the 5 participants did use JavaDoc within the exercises, so a small majority of the participants. However, all the participants mentioned that they knew how to use JavaDoc and several participants explicitly mentioned a the second question that they want to use it in the future (participants 1 and 2) and want to learn more about it for larger projects (participant 2). Therefore, even though JavaDoc was not used that much in the interview, all participants know how to use it and will probably use it in the future and so, the second learning goal is also achieved.

Only participants 1 and 2 used the debugger in the exercises. However, participant 3 mentioned that s(he) could not get the debugger working

anymore after the interview (the necessary windows for debugging did not appear as usual). Participant 4 did not use the debugger personally, but the participants programming partner did, so they actually did use the debugger in the exercise. Participant 5 did not need to use the debugger, because the exercise was not so difficult for that participant. Nonetheless, all the participants mentioned that they knew how to use the debugger and participant 3 mentioned that s(he) will actually use the debugger if possible. Therefore, we already expected from the first interview that the learning goal related to the debugger was achieved, but these results confirm it as well.

Another observation is that the participants that used JavaDoc in the exercises did not use the debugger, but the participants that used the debugger, did not use JavaDoc. We have no explanation for this, so we think this is simply a coincidence.

## 7.4 Combining the results

The pros could directly be used to answer the first research subquestion and the mentioned and found improvements could be mentioned in the discussion. From the first interview was already clear that the learning goals regarding the systematic debugging procedure and the systematic evaluation of the run time error were achieved.

From the pre-test became clear that most participants did not use JavaDoc inside the IDE. The first interviews indicated that most participants did not use JavaDoc, but from the second interviews became clear that several participants used it in practice and all participants knew how to use it. Therefore, the learning goal regarding JavaDoc was also achieved, because participants did not use it before the presentation, but did use it and knew how to use it after the lecture.

From the pre-test became also clear that most participants did not use the debugger while debugging. However, not only did a majority of the participants use the debugger in the first interview, but several participants used it in practice as well and all participants knew how to use it, as indicated by the second interviews. So, the learning goal related to the debugger was achieved as well, because similar to JavaDoc, most participants did not use the debugger before the lecture, but used it after the lecture.

To conclude, all the set learning goals were achieved of the designed lecture and thus, the lecture had its desired and intended effect on the students.

## Chapter 8

# Conclusions

### 8.1 Research subquestions

*What are elements of an effective learning activity about debugging in a programming course?*

In our lecture, we taught the systematic debugging procedure made by Michaeli and Romeike and several debugging strategies that support this debugging procedure, namely using JavaDoc, systematically evaluating a run time error and debugging with the debugger. The learning goals were related to these:

- Students apply the systematic debugging procedure of Michaeli and Romeike when debugging
- Students know about the JavaDoc inside NetBeans
- Students systematically evaluate a run time error to determine the relevant lines in the code and solve the bug
- Students understand the basic functionality of the debugger

These debugging strategies were specifically chosen, because these would solve the most frequently occurring problems for students. As a result, the taught debugging strategies were useful and applicable for the students. In the interviews, some students mentioned that these debugging strategies were very useful and they also applied them in practice, which was the aforementioned aim. Therefore, teaching useful and practical debugging procedures and strategies is an element of an effective learning activity.

Moreover, the integration of direct instruction was also an important element of an effective learning activity. In our lecture, this was primarily done by following the steps of direct instruction and these were supported by a quiz with short exercises, several demos and a debugging exercise at



the end. This led to a very interactive presentation, which should lead to a better understanding of the subjects for the students and other beneficial outcomes for the education. Students specifically mentioned these aspects in the interviews as well; especially the short exercises between the explanations and the interaction provided with the online quiz were mentioned a few times. Moreover, the demos were also appreciated by the students and, according to them, had more impact than several code snippets presented on slides. Therefore, the students were enthusiastic and positive about the integration of direct instruction and the corresponding elements, such as the quiz and the demos.

The lecture took around 45 minutes and as a result, this learning activity could easily be combined with a small tutorial of the course Object Oriented Programming.

*How can the students' resulting debugging skills be characterised?*

After the learning activity, the students used the presented debugging procedure and the systematic evaluation of a run time error while solving the debugging exercise in the first interview. Moreover, not only did the students understand the basics of the debugger, but they actually applied the debugger in practice. The students know about JavaDoc and know how to use it, even though they did not use it a lot yet, but they plan on using it in the future when working on larger and more complex projects. The pre-test indicates that students did not use these debugging strategies before the lecture, so, these observations are dedicated to the learning activity. Therefore, the characteristics of the students' debugging skills after the learning activity were not only the specified learning goals of the lecture, but several students also used JavaDoc and the debugger after the lecture.

## 8.2 Research question

*How can we effectively teach debugging within a programming course?*

Our designed learning activity was effective, as all the learning goals were achieved afterwards. Therefore, this learning activity could be used as teaching material or could be used by other teachers as a stepping stone for their own lecture. It is also convenient that the learning activity takes only 45 minutes, because teachers have limited time left and it could be easily combined with another smaller lecture.

However, it is also possible that teachers want to design their own material from scratch and in that case, it is very useful to integrate the following elements, because these increase the effectiveness of the lecture. One important element of an effective learning activity about debugging is that the

taught debugging strategies and procedures are useful and practical for the students. The taught debugging strategies and procedures in our designed lecture were chosen based on this condition and this was also specifically mentioned by the students; they indicated that they really appreciated that these debugging strategies were useful and very applicable in practice.

Another important element is the integration of direct instruction, especially if it should be taught in one lecture or less. This consists of a few steps, but the most important and appreciated aspects were short exercises in an online quiz between the different subjects, the demos inside a familiar IDE and a debugging exercise to practice with afterwards. This lead to a very interactive learning activity which was helpful and appreciated as indicated by students. Therefore, we advice that these elements are applied in any (programming) course, because it results in beneficial outcomes for the education.

## Chapter 9

# Discussion

In the discussion, we will firstly reflect on our findings and our used methods. Afterwards, we discuss how our findings could be used in practice and lastly, we discuss ideas for future research.

### 9.1 Reflection on findings

The findings we got from the interviews were expected, because integrating direct instruction was regarded as a very effective teaching method and thus, the effects of the lecture could be predicted. We followed the necessary steps for the integration of direct instruction into the lecture and therefore, we got the related benefits of it and so, it was very likely that the learning goals were actually met. Moreover, the debugging strategies that were taught in the lecture were probably useful and practical for the students, because these solved the most frequently occurring bugs encountered by students. Therefore, it was expected that the students were interested in them, because these strategies really helped them, and that the students actually used, or planned to use them, in practice, which was indicated by the interviews. In sum, the students were so positive about this lecture and learned several useful concepts for it, that it would be an idea to give this lecture within this course every year from now on.

A lot of our assumptions in this research were based on other findings of other literature mentioned in the background. For example, the presented debugging procedure was designed and already tested by Michaeli and Romeike and the selection of the presented debugging strategies was based on the most frequently occurring problems described by Altadmri and Brown (2015) and Böttcher et al. (2016). Moreover, the steps of direct instruction and its benefits described in different papers were the main starting points for designing the lecture. So, we used these assumptions to design our learning activity and we proved these assumptions as well, because the findings described in these papers were also recognised in the

findings of the interviews. However, the debugging procedure of Michaeli and Romeike was taught and tested in a K12 classroom environment, while we taught it to first year students of a university, so the environment was quite different. Nonetheless, the debugging procedure was effective as well in this environment and therefore, we tested it in a different environment and that was also effective.

## 9.2 Reflection on methods

Many researchers described in the background, used a debugging exercise to evaluate their learning activity, because it turned out to be a good method to see how the debugging skills of the participants were affected and that became also clear in our research. However, we also did another interview to see if the participants used the presented debugging strategies in practice, which was not used in the other researches. We think that such an interview is really useful, and maybe even necessary, to see the real effect of the learning activity on the students by evaluating how they applied it in practice. It was also an opportunity to evaluate the learning goals in more detail if that was not completely clear after the first interview. Therefore, we advice future researches to have such an evaluation instead of only a debugging exercise.

Even though this research indicates that our designed teaching material about debugging was successful, helpful and appreciated by students, the circumstances of the research were not ideal and could have influenced the outcome of this research a bit in a positive or negative way. Several of these factors are mentioned below:

### **Number of interviewees**

We interviewed 5 students for the analysis of the lecture and this number could be relatively low. The circumstances at that time ensured that it was relatively difficult to find participants for the interviews and we had several time constraints such that we did not had the time to interview more students. In a future research, however, it would be useful to have more students that participate, because that gives a better picture of the effects of the lecture.

### **Biased group of participants in the pre-test**

The pre-test was taken at the designed lecture. The students that were present at that lecture were probably the students that have more difficulty with programming, because the better programmers would not participate in that lecture. These better programmers would probably use the debugger more often than the present programmers. So, the debugger could be used more often by students before the lecture than the pre-test actually

indicates, because the participants in the pre-test could have been biased.

### **Biased results of the first interviews**

In the first interviews, the students were asked to solve a debugging exercise. These interviews were held after the lecture and all the students that participated had seen that lecture as well. This could result in biased results, because students could think that it would be "correct" to apply the presented strategies and therefore, use it in this interview while they did not use it normally which would result in observations that don't reflect the reality. However, this has been partially resolved by also asking the students if they applied the strategies in practice in a second interview, because these results would not be biased as easily as with the interviews.

### **Following programming exercises**

The second interviews investigated whether the students applied the taught debugging strategies in practice within the programming exercises of the course associated with the given lecture. These results could have been biased, because the programming exercises of these weeks were about concurrency in programs. The bugs that usually occur with concurrency problems are often not solved with the debugger and other debugging strategies that were presented, because the occurring errors are quite different. Therefore, it was possible that the students did not use particular debugging strategies, because they did not have the opportunity to do so. However, we partially resolved this by also asking the students if they knew how to use or planned to use a particular strategy when they did not apply it in the programming exercises.

## **9.3 Implications for practice**

As mentioned in the introduction, the problem on which this research is based is the underrepresentation of teaching debugging in education. One reason for this problem was that teachers indicated that there was not enough teaching material to teach debugging to their students. In this research, we created such material and also tested it to verify that it was helpful and successful for the students. Therefore, we solved a really small portion of a relatively big problem, namely for the students of the year 2019-2020 of the course Object Oriented Programming at the Radboud University. The created material can be used in the coming years at this course or could be directly used by other universities and schools.

The designed teaching material could be the stepping stone for other researchers to tackle this huge problem to eventually solve this worldwide underrepresentation of debugging in education. Several ideas are given in the next section.

## 9.4 Future research

One obvious example of future research would be the next iteration of this research, which consists of processing the potential improvements into the lecture: all the learning goals were achieved, but if one explanation of a debugging strategy could be improved, than it should be JavaDoc. The reason for this is that JavaDoc is the only learning goal of which is was not completely clear after the first interview that it was achieved. Another interesting expansion of the lecture could be an explanation about the usage of the tests and tips on writing effective and efficient tests. That combined with the tips for debugging will probably result in a really useful lecture for the students. Another minor improvement would be to take a closer look at the questions inside quizzes to make sure that it is clear for everyone what to answer and that the answers are not ambiguous. Also make sure that the students can see the intermediate results and answers of the quiz. These improvements could be integrated into the lecture and the aforementioned factors could be solved and afterwards, the effect of the lecture could be tested again to see if the effect after the changes. Additionally, it is possible to have another iteration to eventually have a "perfect" lecture about debugging, which would be a huge step against the problem about debugging in education.

Another idea for future research would be to teach the designed lecture at another school or university and test if the same results are found there. It is also possible to test the effect of the lecture more thoroughly. This could be done by replacing the current pre-test, which was a small questionnaire, by a debugging exercise as well and comparing the strategies applied by students before and after the lecture. It could also be possible to have a control group as well by observing students that did follow the lecture and did not follow the lecture to see the difference.

Moreover, it is also possible to make a more "personalised" lecture for the students. The choice of the presented debugging strategies used in this presentation were based on articles written by other researchers. The problems that they identified as most frequently occurring are not necessarily the most frequently occurring problems everywhere. The problems that students could be facing at a university could be unique to that university. To help those students the most, it could be possible to firstly investigate with which problems those students are actually struggling with and base the presented debugging strategies on these results to have the most helpful lecture possible.

# References

- Allwood, C. M., & Björhag, C.-G. (1991). Training of pascal novices' error handling ability. *Acta Psychologica*, 78(1), 137 - 150. Retrieved from <http://www.sciencedirect.com/science/article/pii/000169189190008N> doi: [https://doi.org/10.1016/0001-6918\(91\)90008-N](https://doi.org/10.1016/0001-6918(91)90008-N)
- Altadmri, A., & Brown, N. C. (2015). 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th acm technical symposium on computer science education* (p. 522–527). New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/2676723.2677258> doi: 10.1145/2676723.2677258
- Binder, C., & Watkins, C. L. (1990). Precision teaching and direct instruction: Measurably superior instructional technology in schools. *Performance Improvement Quarterly*, 3(4), 74-96. Retrieved from <https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1937-8327.1990.tb00478.x> doi: 10.1111/j.1937-8327.1990.tb00478.x
- Böttcher, A., Thurner, V., Schlierkamp, K., & Zehetmeier, D. (2016, Oct). Debugging students' debugging process. In *2016 ieee frontiers in education conference (fie)* (p. 1-7). doi: 10.1109/FIE.2016.7757447
- Gauss, E. J. (1982, November). The “wolf fence” algorithm for debugging. *Commun. ACM*, 25(11), 780. Retrieved from <https://doi.org/10.1145/358690.358695> doi: 10.1145/358690.358695
- Gersten, R., & Keating, T. (1987, 01). Long-term benefits from direct instruction. *Educational Leadership*.
- Katz, I., & Anderson, J. (1989, 01). Debugging: An analysis of bug-location strategies. *Human-Computer Interaction*, 3, 351-399. doi: 10.1207/s15327051hci0304.2
- Michaeli, T., & Romeike, R. (2019, April). Current status and perspectives of debugging in the k12 classroom: A qualitative study. In *2019 ieee global engineering education conference (educon)* (p. 1030-1038). doi: 10.1109/EDUCON.2019.8725282
- Michaeli, T., & Romeike, R. (2019, October). Improving debugging skills in the classroom: The effects of teaching a systematic debugging pro-

- cess. In *Proceedings of the 14th workshop in primary and secondary computing education* (pp. 15:1–15:7). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/3361721.3361724> doi: 10.1145/3361721.3361724
- Perscheid, M., Siegmund, B., Taeumel, M., & Hirschfeld, R. (2017, March). Studying the advancement in debugging practice of professional software developers. *Software Quality Journal*, 25(1), 83–110. Retrieved from <https://doi.org/10.1007/s11219-015-9294-2> doi: 10.1007/s11219-015-9294-2
- Renard, L. (2019). *Direct instruction - a practical guide to effective teaching*. Retrieved 2020-03-02, from <http://https://www.bookwidgets.com/blog/2019/03/direct-instruction-a-practical-guide-to-effective-teaching>
- van den Akker, J., McKenney, S., Nieveen, N., & Gravemeijer, K. (2006). Introducing educational design research. In J. van den Akker, K. Gravemeijer, S. McKenney, & N. Nieveen (Eds.), *Educational design research* (pp. 3–7). United Kingdom: Routledge.
- Watkins, C. L. (1988). Project follow through: A story of the identification and neglect of effective instruction. *Youth Policy*, 10(1).
- Zeller, A. (2009). *Why programs fail : a guide to systematic debugging*. San Francisco, Calif. Oxford: Morgan Kaufmann Elsevier Science distributor.



## Appendix A

### Figures

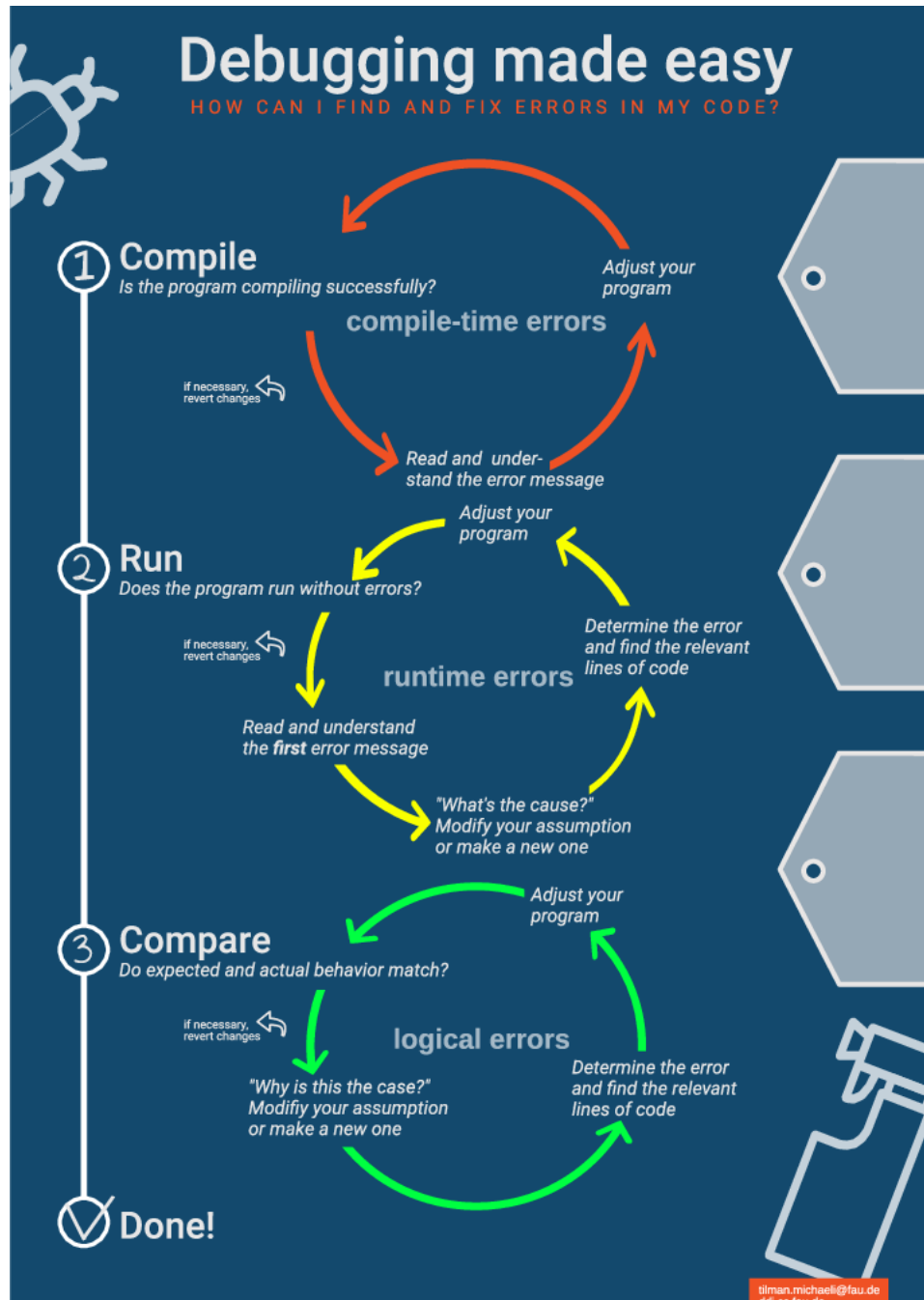


Figure A.1: The schematic representation of the debugging process created by Michaeli and Romeike (2019)

‘(1) Start to correct the bugs in the order in which they are marked in the program (i.e. not in the order they are given in the compiler’s “error list”). Try not to deal with too many bugs at the same time; preferably deal with only one bug at a time (you might get rid of a few sequel bugs at the same time). When you start to correct a bug, first read the error message dealing with the bug and explain the content of the error message.

(2) When you have understood the meaning of the error message for a certain bug but you do not immediately realize what the error is, bring up the program on the screen. Next try to localize the area in the program which the compiler might have indicated. Then look for syntax errors if the text of the error message gives you reason to suspect this. Examples of syntax errors are: omission of a semicolon (;), one semicolon too many, and the use of loop-constructions which are not grammatically correct.

If the error message indicates other types of bugs try to trace these by carefully following the flow of information from the beginning of the program. If this seems completely unnecessary, start to follow the flow of information a little above the place in the program suggested by the compiler as the place of the bug. For each program line that you read in the program state aloud what is carried out in that line. Then consider if it fits in with what the program did earlier and if it fits with what you planned that the program should do.

(3) Before you correct a bug, carefully make it clear for yourself what the nature of the bug is. Before you carry out a correction you should make yourself aware of what consequences the change will have on the rest of your program. Also make sure your change does not violate any of the assumptions which the program is dependent on.

(4) If you have corrected a bug which, in turn, might have given rise to other bugs you should immediately investigate the effect of your correction by compiling or running the program in order to find out if your change has had the intended effect.’

Figure A.2: The instructions presented in the document of Allwood and Björhag (1991)

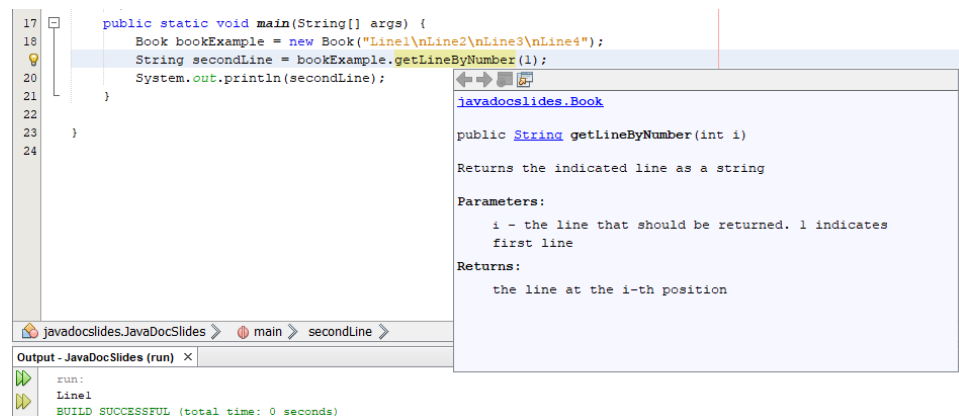


Figure A.3: The picture used in question 6 of the Socratic Quiz

```
run:
[ ] Exception in thread "main" java.lang.ArithmeticException: / by zero
  |   at runtimequiz.RunTimeQuiz.g(RunTimeQuiz.java:27)
  |   at runtimequiz.RunTimeQuiz.f(RunTimeQuiz.java:22)
  |   at runtimequiz.RunTimeQuiz.main(RunTimeQuiz.java:18)
C:\Users\ruben\AppData\Local\NetBeans\Cache\8.2\executor-snippets\run.xml:53: Java returned: 1
BUILD FAILED (total time: 0 seconds)
```

Figure A.4: The picture used in question 7 of the Socratic Quiz

# Appendix B

## Code Fragments

## B.1 Debugging exercise after presentation: Calculate average price of a list of fruits

```
1 package debuggingexercisel1;
2
3 public class DebuggingExercise1 {
4
5     public static void main(String[] args) {
6         Fruit[] list1 = {new Apple(), new Banana()};
7         Fruit[] list2 = {new Apple(), new Banana(), new Pear(),
8             new Orange(), new Orange(), new Pear()};
9         Fruit[] list3 = {};
10        Fruit[] list4 = {new Apple(), new Orange()};
11        int resList1 = calcAveragePriceOfFruit(list1);
12        int resList2 = calcAveragePriceOfFruit(list2);
13        int resList3 = calcAveragePriceOfFruit(list3);
14        int resList4 = calcAveragePriceOfFruit(list4);
15        System.out.println("The average of list 1 should be
16        5.0, current value = " + resList1);
17        System.out.println("The average of list 2 should be
18        7.0, current value = " + resList2);
19        System.out.println("The average of list 3 should be
20        0.0, current value = " + resList3); //This is debatable,
21        but in this exercise, we assume that an average of 0
22        elements is 0
23        System.out.println("The average of list 4 should be
24        6.5, current value = " + resList4);
25
26    }
27
28    public static double calcAveragePriceOfFruit(Fruit[] fruits
29    ){
30        int sum = 0;
31        for(int i = 1; i <= fruits.length; i++){
32            sum += fruits[i].getPrice();
33        }
34        double result = sum/fruits.length;
35        return result;
36    }
37 }
```

Listing B.1: The main class (DebuggingExercise1.java)

```
1 package debuggingexercisel1;
2
3 public interface Fruit {
4     abstract public int getPrice();
5     abstract public int getWeight();
6 }
```

Listing B.2: Fruit.java

```
1 package debuggingexercisel1;
```

```

2
3 public class Apple implements Fruit {
4     int price = 6;
5     int weight = 3;
6
7     @Override
8     public int getWeight(){
9         return price;
10    }
11
12    @Override
13    public int getPrice(){
14        return weight;
15    }
16 }

```

Listing B.3: Apple.java

```

1 package debuggingexercise1;
2
3 public class Banana implements Fruit {
4     int price = 4;
5     int weight = 4;
6
7     @Override
8     public int getWeight(){
9         return weight;
10    }
11
12    @Override
13    public int getPrice(){
14        return price
15    }
16 }

```

Listing B.4: Banana.java

```

1 package debuggingexercise1;
2
3 public class Orange implements Fruit {
4     int price = 7;
5     int weight == 3;
6
7     @Override
8     public int getWeight[]{
9         return weight;
10    }
11
12    @Override
13    public int getPrice(){
14        return price;
15    }
16 }

```

Listing B.5: Orange.java

```

1 package debuggingexercise1;
2
3 public class Pear implements Fruit {
4     int price = 9;
5     int weight = 4;
6
7     @Override
8     public int getWeight(){
9         return weight;
10    }
11
12    @Override
13    public int getPrice(){
14        return -price;
15    }
16 }

```

Listing B.6: Pear.java

## B.2 Debugging exercise of the interview: Calculate median of a list of integers

```

1 package debuggingexercise3;
2
3 import java.util.ArrayList;
4 import java.util.Arrays;
5
6 public class DebuggingExercise3 {
7
8     public static void main(String[] args) {
9         int[] list1 = {1,2,3};
10        int[] list2 = {3,2,5,3,2,5};
11        int[] list3 = {1,8,4,8,45,8,4,976,234,867800,34,9};
12        int[] list4 = {5,7,3,5,8,9,3,4,7,4,8};
13        int[] resultList1 = calcMedian(list1);
14        int[] resultList2 = calcMedian(list2);
15        int[] resultList3 = calcMedian(list3);
16        int[] resultList4 = calcMedian(list4);
17        System.out.println("Result of list1 should be [2],
18        current value is " + Arrays.toString(resultList1));
19        System.out.println("Result of list2 should be [3,3],
20        current value is " + Arrays.toString(resultList2));
21        System.out.println("Result of list3 should be [8,9],
22        current value is " + Arrays.toString(resultList3));
23        System.out.println("Result of list4 should be [5],
24        current value is " + Arrays.toString(resultList4));
25    }
26
27    /**
28     * Calculates the median of the given array
29     * @param numbers The numbers of which the median should be
30     * found
31     * @return An array with one or both medians
32     */
33 }

```



```

27     */
28     public static int[] calcMedian(int[] numbers){
29         ArrayList<Integer> sortedNumbers = sortNumbers(numbers)
30     ;
31         if(numbers.length % 2 == 1){
32             int[] result = {sortedNumbers[numbers.length/2+1],
33 sortedNumbers[numbers.length/2]};
34             return result;
35         }
36         int[] result = {sortedNumbers[numbers.length/2]};
37         return result;
38     }
39     /**
40     * Sorts a given array through selection sort
41     * @param list The array to be sorted
42     * @return A sorted array
43     */
44     public static int[] sortNumbers(int[] list){
45         for(int i = 0; i <= list.length; i++){
46             int index = findIndexWithSmallestNumber(list, i,
47 list.length);
48             // Switch the found index with i, such that 0 till
49 i is already sorted
50             int x = list[index];
51             list[index] = list[i];
52             list[i] = x;
53         }
54         return list;
55     }
56     /**
57     * Finds the index of the smallest number within the given
58 bounds in an array
59     * @param list The given array
60     * @param l The left bound (inclusive)
61     * @param r The right bound (exclusive)
62     * @return The index with the smallest number
63     */
64     public static int findIndexWithSmallestNumber(int[] list,
65 int l, int r){
66         int result = l;
67         for(int i = l + 1; i < r; i++){
68             if(list[i] < list[result]){
69                 result = i;
70             }
71         }
72         return result;
73     }
74 }

```

Listing B.7: The main class (DebuggingExercise3.java)

## B.3 JavaDoc demo used in the presentation

```

1 package javadocdemo;
2
3 public class JavaDocDemo {
4
5     public static void main(String[] args) {
6         printTrimmedString("    Hello, world!    ");
7     }
8
9     public static void printTrimmedString(String text){
10        text.trim();
11        System.out.println("(" + text + ")");
12    }
13
14 }

```

Listing B.8: The main class (JavaDocDemo.java)

## B.4 Run Time error demo used in the presentation

```

1 package runtime demo;
2
3 import java.util.ArrayList;
4
5 public class RunTimeDemo {
6
7     public static void main(String[] args) {
8         ArrayList<Integer> x = new ArrayList();
9         for(int i = 0; i < 10; i++){
10             x.add(null);
11         }
12         for(int i = 1; i < x.size(); i++){
13             x.set(i, i);
14         }
15         for(int number : x){
16             number++;
17         }
18         System.out.println(x);
19     }
20
21 }

```

Listing B.9: The main class (RunTimeDemo.java)

## B.5 The first debugger demo used in the presentation

```

1 package debuggerdemo1;
2
3 public class DebuggerDemo1 {
4
5     public static void main(String[] args) {

```

```

6      int[] input = {1,2,3};
7      double avg = findAverage(input);
8      System.out.println("The average is " + avg);
9  }
10
11     private static double findAverage(int[] input) {
12         double result = 0;
13         for (int s : input) {
14             result += s;
15         }
16         return result;
17     }
18 }
19 }

```

Listing B.10: The main class (DebuggerDemo1.java)

## B.6 The second debugger demo used in the presentation

```

1 package debuggerdemo2;
2
3 public class DebuggerDemo2 {
4
5     public static void main(String[] args) {
6         int x = 2;
7         int y = 3;
8         int z = 4;
9
10        String s1 = "Hello, ";
11        String s2 = "world!";
12        s1 = s1.toLowerCase().concat(s2);
13
14        f(s1);
15        f(s1);
16
17        s1 = "Hello, ";
18        s2 = "world!";
19        s1 = s1.concat(s2);
20    }
21
22    public static void f(String s){
23        s = s.toLowerCase();
24        s = s.toUpperCase();
25        boolean b = s.contains("H");
26        s = s.toLowerCase();
27        s = s.toUpperCase();
28    }
29 }
30 }

```

Listing B.11: The main class (DebuggerDemo2.java)

# Appendix C

## Documents

## C.1 Pdf answer debugging exercise after presentation

### Answers Debugging exercise 1

Ruben Holubek s1006591

March 2020

- Compile time errors
  1. In Banana.java, in line 14:  
Missing semicolon.
  2. In Orange.java, in line 5:  
== should be a single =. Now a comparison is made instead of a declaration.
  3. In Orange.java, in line 8:  
Wrong brackets, should be () instead of [].
  4. In DebuggingExercise1.java, in lines 10-13:  
The types should be double and not int.
- Run time errors
  1. In DebuggingExercise1.java, in line 23:  
The  $\leq$  in the loop should be a  $<$ . Now an indexOutOfBounds Error occurs.
  2. In DebuggingExercise1.java, in line 26:  
The case where the list is empty should be handled separately, because otherwise a division by 0 occurs. This could be done by a separate if statement to catch this case.
- Logic errors
  1. In Apple.java, in line 9 and 14:  
The price and weight are switched in the getWeight() and getPrice() methods.
  2. In Pear.java, in line 14:  
The minus sign should be removed.
  3. In DebuggingExercise1.java, in line 23:  
The for loop should start at 0 and not 1. Now the first element is always skipped.
  4. In DebuggingExercise1.java, in line 26:  
The division should be converted to a double by typing (double) before it. Currently, 2 integers are divided and the result will be an integer as well, which is then converted to a double. But by converting the type before the assignment, the double is correctly stored.

## C.2 Pdf answer debugging exercise used in the interview

### Answers Debugging exercise 3

Ruben Holubek s1006591

March 2020

- Compile time errors
  1. In line 29:  
ArrayList<Integer> should be an int[].
- Run time errors
  1. In line 44:  
The  $\leq$  should be a  $<$  in the loop. Otherwise an indexOutOfBoundsException occurs.
- Logic errors
  1. In line 30:  
The 1 should be replaced with a 0. Otherwise the wrong case is caught.
  2. In line 31:  
The numbers.length/2+1 should be a numbers.length/2-1. Otherwise the wrong indices are returned.
  3. In line 62:  
The  $>$  should be a  $<$  or  $\leq$ . Otherwise the index with the largest number is returned and so, the list will be sorted the other way around.

## C.3 Schema used for the interviews

Interview person x

1. In line 29 (Compile):  
ArrayList<Integer> should be an int[].  
How fixed:
2. In line 44 (Run):  
The <= should be a < in the loop.  
How fixed:
3. In line 30 (Logic):  
The 1 should be replaced with a 0.  
How fixed:
4. In line 31 (Logic):  
The numbers.length/2+1 should be a numbers.lengt/2-1  
How fixed:
5. In line 62 (Logic)::  
The > should be a < or <=.  
How fixed:

Order in which the bugs were fixed:

Overall strategy:

Good points:

Improvements:

Other notes:

## C.4 Socrative Quiz used in the presentation



Name \_\_\_\_\_

Date \_\_\_\_\_

# Debugging Teaching Intervention

Score \_\_\_\_\_

1. Do you use the following strategy when debugging? Searching on Stack Overflow

- ☐ A Yes
- ☐ B No
- ☐ C No opinion

2. Do you use the following strategy when debugging? Looking at documentation online

- ☐ A Yes
- ☐ B No
- ☐ C No opinion

3. Do you use the following strategy when debugging? Looking at documentation inside the IDE (NetBeans)

- ☐ A Yes
- ☐ B No
- ☐ C No opinion

4. Do you use the following strategy when debugging? Print statements to see the values of the variables

- ☐ A Yes
- ☐ B No
- ☐ C No opinion

5. Do you use the following strategy when debugging? The debugger

- ☐ A Yes
- ☐ B No
- ☐ C No opinion



6.

```

17 public static void main(String[] args) {
18     Book bookExample = new Book("Line1\nLine2\nLine3");
19     String secondLine = bookExample.getLineByNumber(1);
20     System.out.println(secondLine);
21 }
22
23
24

```

Output: JavaDocSlides (run) X

main > secondLine >

Line1

BUILD SUCCESSFUL (total time: 0 seconds)

JavaDocSlides.Book

public String getLineByNumber(int i)

Returns the indicated line as a string

Parameters:

i - the line that should be returned. 1 indicates first line

Returns:

the line at the i-th position

SecondLine should be "Line2", but it prints "Line1". How should this bug probably be solved?

- (A) Edit the function getLineByNumber()
- (B) Edit the given argument
- (C) Edit the print System.out.println() statement

7.

```

run:
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at runtimequiz.RuntimeQuiz.g(RuntimeQuiz.java:27)
    at runtimequiz.RuntimeQuiz.f(RuntimeQuiz.java:22)
    at runtimequiz.RuntimeQuiz.main(RuntimeQuiz.java:18)
C:\Users\ruben\AppData\Local\NetBeans\Cache\8.2\executor-snippets\run.xml:53: Java returned: 1
BUILD FAILED (total time: 0 seconds)

```

Given this Run Time error, what is probably wrong?

- (A) A null object at line 18
- (B) A null object at line 27
- (C) A bug that has something to do with indices and arrays/lists at line 18
- (D) A bug that has something to do with indices and arrays/lists at line 27
- (E) A bug in a mathematical statement at line 18
- (F) A bug in a mathematical statement at line 27

## C.5 Filled in interview schemes

Interview person 1

1. In line 29 (Compile):  
ArrayList<Integer> should be an int[].  
How fixed:  
Looked at the line number given by the compiler.  
Determined it was a wrong type.  
Used the quick solution provided by the IDE.
2. In line 44 (Run):  
The <= should be a < in the loop.  
How fixed:  
After the run time error, looked at the provided line and immediately saw that it should be a smaller than, because the interviewee made that mistakes often in practice.
3. In line 30 (Logic):  
The 1 should be replaced with a 0.  
How fixed:  
Used the debugger to see how the variables changed step-by-step. The interviewee saw that the wrong case was executed and switched them to fix the bug, after explaining the function to herself.
4. In line 31 (Logic):  
The numbers.length/2+1 should be a numbers.lengt/2-1  
How fixed:  
Used the debugger to see how the variables changed step-by-step. The interviewee saw that the +1 was incorrect and removed it. After using the debugger once more, she saw that it still was incorrect and added -1 to fix the bug.
5. In line 62 (Logic):  
The > should be a < or <=.  
How fixed:  
Used the debugger in the corresponding function to see how the variables changed step-by-step. Determined that the function did not work correctly and explained the function to herself and saw that the sign was wrong.

Order in which the bugs were fixed:

1-2-5-3-4

Overall strategy:

1. Ran program -> Compile time error
2. Fixed bug 1
3. Ran program -> Run time error
4. Fixed bug 2
5. Ran program -> Test case 1 was incorrect
6. Inspected findIndexWithSmallestNumber with test case 1 with debugger -> incorrect result
7. Fixed bug 5
8. Ran program -> Test case 1 was incorrect

9. Inspected findIndexWithSmallestNumber with test case 1 with debugger -> correct result
10. Inspected sortNumbers with test case 1 with debugger-> correct result
11. Inspected calcMedian with test case 1 with debugger -> incorrect result
12. Incorrectly fixed bug 3, but test case 1 was correct
13. Ran program -> Test case 1 was correct
14. Ran program -> Test case 2 was incorrect
15. Inspected sortNumbers with test case 2 with debugger-> correct result
16. Inspected calcMedian with test case 2 with debugger-> incorrect result
17. Fixed bug 3
18. Ran program -> Test case 2 was correct
19. Ran program -> Test case 3 was incorrect
20. Inspected sortNumbers with test case 2 with debugger-> incorrect result
21. Fixed bug 4
22. Ran program -> Test case 3 was correct
23. Ran program -> Test case 4 was correct
24. Ran program -> All test cases were correct

Good points:

Demo's were really useful, helped to see how it works in practice instead on slides.

The explanations were clear.

Improvements:

The full picture of the debugging strategy was rather small.

Other notes:

Was thinking out loud

Never used debugger, actually did after the intervention and was really useful

## Interview person 2

1. In line 29 (Compile):  
ArrayList<Integer> should be an int[].  
How fixed:  
Saw error in the IDE, checked the import and used the shortcut for more information.  
Afterwards used the quick fix given by the IDE.
2. In line 44 (Run):  
The <= should be a < in the loop.  
How fixed:  
Saw the run time error and inspected the provided line and immediately fixed it by looking at it.
3. In line 30 (Logic):  
The 1 should be replaced with a 0.  
How fixed:  
Looked at all the test cases and saw that the number of the returned integers was incorrect and therefore thought that it had to do with the if statement. Used the debugger to see the execution step-by-step and fixed the error.
4. In line 31 (Logic):  
The numbers.length/2+1 should be a numbers.length/2-1  
How fixed:  
With the debugger, the interviewee knew that it had to do with calcMedian and with the help with the debugger and test case 2, after a while the interviewee saw that the incorrect indices were used and fixed the bug.
5. In line 62 (Logic):  
The > should be a < or <=.  
How fixed:  
Inspected the findIndexWithSmallestNumber() with the debugger and saw that the result was incorrect of case 2 and fixed the bug after inspecting it step-by-step.

Order in which the bugs were fixed: 1-2-3-5-4

## Overall strategy:

1. Looked at the compile time error provided by the IDE
2. Fixed bug 1
3. Ran program -> Run time error
4. Fixed bug 2
5. Ran program -> All test cases incorrect, but in particular the length of the results were incorrect
6. Inspected calcMedian with test case 1 with debugger -> incorrect result
7. Fixed bug 3
8. Ran program -> Test cases 2 & 3 incorrect
9. Inspected sortNumbers with test case 1 with debugger -> incorrect result
10. Inspected findIndexWithSmallestNumber with test case 1 with debugger -> incorrect result

11. Fixed bug 5
12. Ran program -> Test cases 2 & 3 incorrect
13. Inspected findIndexWithSmallestNumber with test case 2 with debugger-> correct result
14. Inspected sortNumbers with test case 2 with debugger -> correct result
15. Inspected calcMedian with test case 2 with debugger -> incorrect result
16. Fixed bug 4
17. Ran program -> All test cases were correct

Good points:

Demo's were really useful, helped to see how it works in practice instead on slides.

The interaction with the students was really useful.

The increasing steps were very helpful (first print statements, than the debugger)

Improvements:

Show the end score of the quiz.

Other notes:

Was thinking out loud

Never used debugger, actually did after the intervention and immediately solved several bugs

### Interview person 3

1. In line 29 (Compile):  
ArrayList<Integer> should be an int[].  
How fixed:  
Looked at the error given by the IDE. Saw that the types were incorrect and changed the types by hand.
2. In line 44 (Run):  
The <= should be a < in the loop.  
How fixed:  
Evaluated the run time error, went to the correct line and immediately changed it.
3. In line 30 (Logic):  
The 1 should be replaced with a 0.  
How fixed:  
Saw that there was a bug associated with the length of the returned object. Therefore, inspected the if statement and saw that the wrong case was applied and fixed it.
4. In line 31 (Logic):  
The numbers.length/2+1 should be a numbers.length/2-1  
How fixed:  
Knew that the bug was in calcMedian and just followed the code step by step in her head. She saw that the incorrect index was returned after using an example and firstly removed the +1, went through hit again, and changed it to a -1.
5. In line 62 (Logic):  
The > should be a < or <=.  
How fixed:  
The interviewee inspected the findIndexWithSmallestNumber with the debugger and saw that the result was incorrect. The interviewee saw the bug relatively quickly and fixed it.

Order in which the bugs were fixed: 1-2-3-5-4

### Overall strategy:

1. Looked at the compile time error provided by the IDE
2. Fixed bug 1
3. Ran program -> Run time error
4. Fixed bug 2
5. Ran program -> All test cases incorrect, but in particular the length of the results were incorrect
6. Looked at calcMedian
7. Fixed bug 3
8. Ran program -> Test cases 2 & 3 incorrect
9. Inspected sortNumbers with test case 2 with debugger -> incorrect result
10. Inspected findIndexWithSmallestNumber with test case 2 with debugger -> incorrect result
11. Fixed bug 5
12. Ran program -> Test cases 2 & 3 incorrect

13. Inspected findIndexWithSmallestNumber with test case 2 with debugger-> correct result
14. Inspected sortNumbers with test case 2 with debugger -> correct result
15. Inspected calcMedian with test case 2 with debugger -> incorrect result
16. Fixed bug 4
17. Ran program -> All test cases were correct

Good points:

Clear explanation.

The images were clear (in particular the commands)

The interaction/feedback and the demos were really nice to see if the students understood the material.

Improvements:

An "in between" option in the quiz (e.g. sometimes instead of only yes and no)

Other notes:

Was thinking out loud

Never used debugger, actually did after the intervention

#### Interview person 4

1. In line 29 (Compile):  
ArrayList<Integer> should be an int[].  
How fixed:  
Looked at error provided by IDE, type was incorrect and fixed it by hand.
2. In line 44 (Run):  
The <= should be a < in the loop.  
How fixed:  
Run time error occurred, looked at the stack trace and the lines and immediately fixed it.
3. In line 30 (Logic):  
The 1 should be replaced with a 0.  
How fixed:  
Saw that the length was incorrect and immediately knew that it had to do with the if statement and fixed it.
4. In line 31 (Logic):  
The numbers.length/2+1 should be a numbers.length/2-1  
How fixed:  
Saw that the output was incorrect and knew it had to be in calcMedian and immediately solved it there.
5. In line 62 (Logic):  
The > should be a < or <=.  
How fixed:  
Looked at the function after determining that it had to be in this function and immediately fixed it.

Order in which the bugs were fixed:

1-2-3-5-4

Overall strategy:

1. Looked at the compile time error provided by the IDE
2. Fixed bug 1
3. Ran program -> Run time error
4. Fixed bug 2
5. Ran program -> All test cases incorrect, but in particular the length of the results were incorrect
6. Looked at calcMedian
7. Fixed bug 3
8. Ran program -> Test cases 2 & 3 incorrect
9. Used print statement to see the outcome of sortNumbers -> Numbers were sorted in the wrong way around
10. Looked at findIndexWithSmallestNumber
11. Fixed bug 5
12. Ran program -> Test cases 2 & 3 incorrect



- 13. Knew the numbers were correctly sorted -> had to be in calcMedian
- 14. Fixed bug 4
- 15. Ran program -> All test cases were correct

Good points:

The interaction was nice, especially the demos and the quiz.  
That kept the students engaged.

Improvements:

-

Other notes:

Prefers print statements if the interviewee only wants to see a specific value , otherwise uses the debugger.  
Was relatively a good programmer and thought this exercise was rather easy and included frequently occurring bugs.  
Was thinking out loud

#### Interview person 5

1. In line 29 (Compile):  
ArrayList<Integer> should be an int[].  
How fixed:  
Looked at error provided by IDE, type was incorrect and fixed it by hand.
2. In line 44 (Run):  
The <= should be a < in the loop.  
How fixed:  
Run time error occurred, looked at the relevant lines and immediately fixed it.
3. In line 30 (Logic):  
The 1 should be replaced with a 0.  
How fixed:  
Saw that the length was incorrect and knew that it had to be in calcMedian. Used the debugger step-by-step to see what went wrong and fixed it.
4. In line 31 (Logic):  
The numbers.length/2+1 should be a numbers.length/2-1  
How fixed:  
Saw that the output was incorrect and knew it had to be in calcMedian with the returned positions. Used an example and saw that the indices were indeed incorrect and corrected them.
5. In line 62 (Logic):  
The > should be a < or <=.  
How fixed:  
Saw that the sortNumbers sorted the other way around and inspected findIndexWithSmallestNumber with the debugger step-by-step and fixed it.

Order in which the bugs were fixed:

1-2-3-4-5

Overall strategy:

1. Looked at the compile time error provided by the IDE
2. Fixed bug 1
3. Ran program -> Run time error
4. Fixed bug 2
5. Ran program -> All test cases incorrect, but in particular the length of the results were incorrect
6. Looked at calcMedian
7. Fixed bug 3
8. Ran program -> Test cases 2 & 3 incorrect, but in particular the returned positions were incorrect
9. Inspected calcMedian with the debugger
10. Fixed bug 4

11. Ran program -> Test case 3 incorrect, in particular the sorting algorithm was probably incorrect
12. Inspected sortNumbers with the debugger -> sorted the other way around
13. Looked at findIndexWithSmallestNumber with the debugger
14. Fixed bug 5
15. Ran program -> All test cases were correct

Good points:

The explanations were really helpful and clear.

Especially the run time error explanations were useful

Improvements:

-

Other notes:

Never used the debugger before, but did after the learning activity

Was thinking out loud

## C.6 Results of the second interviews

### Participant 1:

Did not use JavaDoc the last weeks since it wasn't necessary for that interviewee, but the interviewee knows how to use it and definitely will use in the future. The interviewee did use the debugger (a lot) and found it a very useful addition to the course.

### Participant 2:

Did not used Javadoc that much but understands the basics and would like to learn more about it when working on bigger projects. The participant did use the debugger a lot and it helped a lot with the exercises.

### Participant 3:

Did use JavaDoc in the exercises, but did not use the debugger, because it did not work anymore in the participants Netbeans IDE after the interview. However, the participant knows how to use the debugger now and will use it in the future.

### Participant 4:

Used JavaDoc extensively, whenever not sure how a certain Object in Java works or when looking for a specific function that's where the participant starts to look it up. The participant did not use the debugger personally, but its programming partner did when they encountered one very elusive bug. Nevertheless, the participant would roughly know how to use the debugger if needed.

### Participant 5:

Did only one java exercise since the first interview, but used Javadoc in that exercise. However, the exercise was not that difficult and so, the participant did not need the debugger, but the participant would know how to use it.

## C.7 Used slides in the presentation

# How to debug

## Strategies and tips to efficiently debug your programs

Ruben Holubek  
Radboud University Nijmegen  
May 18, 2020

## Table of contents

- Introduction
- A systematic debugging procedure
- JavaDoc
- Evaluating a run time error
- The debugger
- Wrapping it up
- Time to practice

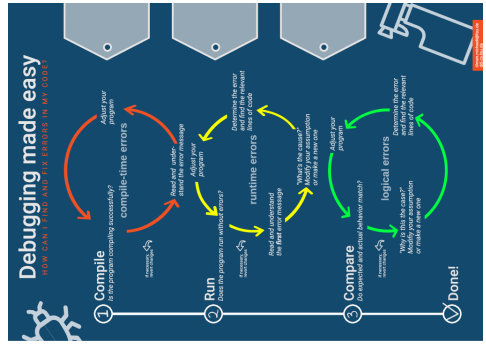
## Table of contents

- Introduction
- A systematic debugging procedure
- JavaDoc
- Evaluating a run time error
- The debugger
- Wrapping it up
- Time to practice

## About me

- Ruben Holubek
- 3rd year bachelor student
- Bachelor thesis
- Teach students how to debug and test the effects

# Overview



Explained tools/concepts:

- JavaDoc (Compile, Run and Compare)
- Evaluating a run time error (Run)
- The debugger (Run and Compare)

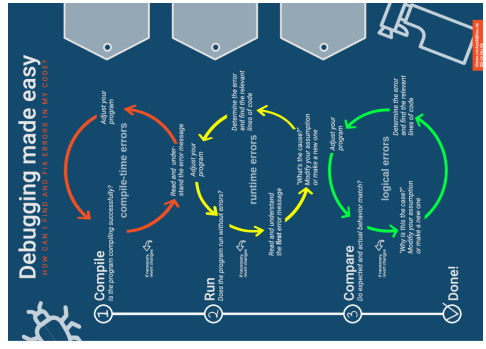
# But first, a short quiz...

- 1 Go to socrative.com
- 2 Press on login
- 3 Login as student
- 4 Room name is HOLUBEK60

# Table of contents

- Introduction
- A systematic debugging procedure
- JavaDoc
- Evaluating a run time error
- The debugger
- Wrapping it up
- Time to practice

# A systematic debugging procedure



- 3 different debugging phases
  - Compile
  - Run
  - Compare
- Used in paper from Michaeli and Romeike





## Table of contents

- Introduction
- A systematic debugging procedure
- JavaDoc
- Evaluating a run time error
- The debugger
- Wrapping it up
- Time to practice

## JavaDoc

- Documentation of code
- Ctrl + Shift + Space to see JavaDoc inside Netbeans!
- Shows information about the arguments, return object, the functionality and more
- Works also for self-written documentation

## Example

```
public static void main(String[] args) {
    String part1 = "Part1&ampnd";
    System.out.println(examp
        java.lang.String
        Book bookExample = new B
        String secondLine = book
        public String concat(String string)
        System.out.println(second
    )
    Concatenates the specified string to the end of this
    string.
    If the length of the argument string is 0, then this String
    object is returned. Otherwise, a String object is returned
    that represents a character sequence that is the
    concatenation of the character sequence represented by
    this String object and the character sequence represented
    by the argument string.
    Examples:
    slides.JavaDocSlides > main > exa
    DocSlides (run) x
```

## Our documentation...

```
public class Book {
    String content;

    public Book(String content){
        this.content = content;
    }

    /**
     * Returns the indicated line as a string
     * @param i the line that should be returned. 1 indicates first line
     * @return the line at the i-th position
     */
    public String getLineByNumber(int i){
        String[] lines = content.split("\\n");
        return lines[i-1];
    }
}
```

## ... also seen in JavaDoc

```
public class JavaDocSlides {  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        String part1 = "Part1and";  
        String example = part1.concat("Part2");  
        System.out.println(example);  
  
        Book bookExample = new Book("Title\\nline3\\nline4");  
        String secondLine = bookExample.getLineByNumber(1);  
        System.out.println(secondLine);  
    }  
}
```

JavaDocSlides.Book

public String getLineByNumber(int i)  
Returns the indicated line as a string  
Parameters:  
i - the line that should be returned. 1 indicates first line  
Returns:  
the line at the i-th position

JavaDocSlides.JavaDocSlides

main

secondLine

part1 - JavaDocSlides (run) X

part1andPart2

example

BUILD SUCCESSFUL (total time: 0 seconds)

## Demo and exercise

- Demo about using JavaDoc
- A question at Socrative

## Table of contents

- Introduction
- A systematic debugging procedure
- JavaDoc
- Evaluating a run time error
- The debugger
- Wrapping it up
- Time to practice

## First an example

```
12 public class RunTimeSlides {  
13     public static void main(String[] args) {  
14         int[] numbers = {1,6,8,10};  
15         int sum = calcSum(numbers);  
16         System.out.println(sum);  
17     }  
18  
19     public static int calcSum(int[] numbers){  
20         int sum = 0;  
21         for(int i = 0; i <= numbers.length; i++){  
22             sum += numbers[i];  
23         }  
24         return sum;  
25     }  
26 }  
27
```

Output X

be-theses - P:\Mijn Documenten\Radboud\jaar\_3\BachelorScriptie\Git folder\ba-theses X RunTimeSlides (run) X

run:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4  
at runTimeSlides.RunTimeSlides.calcSum(RunTimeSlides.java:22)  
at runTimeSlides.RunTimeSlides.main(RunTimeSlides.java:15)  
C:\Users\Ruben\AppData\Local\NetBeans\Cache\8.2\executor-nippeta\run.xml:53: Java returned: 1  
BUILD FAILED (total time: 0 seconds)



**Null Pointer Exception:**

**Explanation:** This is thrown when Java encounters a null reference when it doesn't expect one.

This usually happens when something hasn't been initialized or instantiated.

**Solution:** Use print statements to determine where it is null (generally in a for loop).

**Arithmetic Exception:**

**Explanation:** This generally happens when you try to divide by zero.

**Solution:** Wrap the code that can divide by zero with try/catch.

**Class Cast Exception:**

**Explanation:** This happens when you try to wrongly cast an object to a particular class. This means that your object is not an instance or a subclass of this class.

**Solution:** Look at your class hierarchy and make sure your subclasses inherit properly.

**Stack Overflow Exception:**

**Explanation:** This happens when Java runs out of it's available memory.

**Solution:** This is generally caused by an infinite loop or infinite recursion, so looking at those loops and recursive calls is a good place to start.

- Demo about parsing a Run Time error
- A question at Socrative

Introduction

A systematic debugging procedure

JavaDoc

Evaluating a run time error

The debugger

Wrapping it up

Time to practice

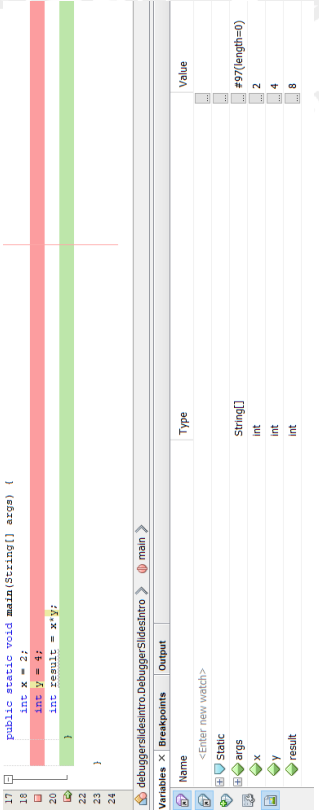
- A tool to see the current state of the program
- Very effective and efficient for finding bugs
- A print statement on steroids
- Available in most IDE's
- So very useful to learn how to use it!
- To start the debugger: Debug → debug project (or Ctrl + F5)

### Example I

```
17 public static void main(String[] args) {
18     int x = 2;
19     int y = 4;
20     int result = x*y;
21 }
22
23 }
```

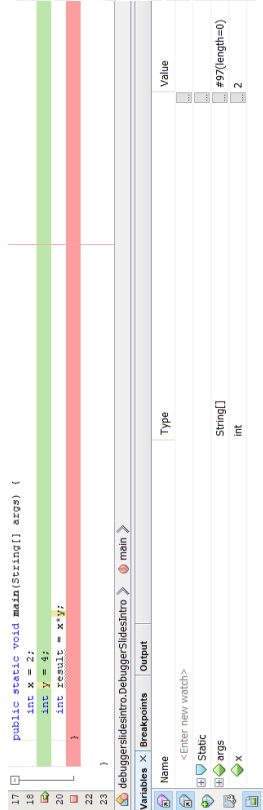
- Program before starting debugger
- Breakpoints on lines 19 and 21
- Breakpoints can be set by clicking on the line numbers

### Example III



- Debugger currently on breakpoint on line 21
- Values of x, y and result can be inspected

### Example II



- Debugger currently on breakpoint on line 19 (Not executed yet!)
- Program is currently paused
- Variables can be inspected
- x has value 2, y and result do not exist yet

### Demo

Demo about the basics of the debugger

## Where to place breakpoints

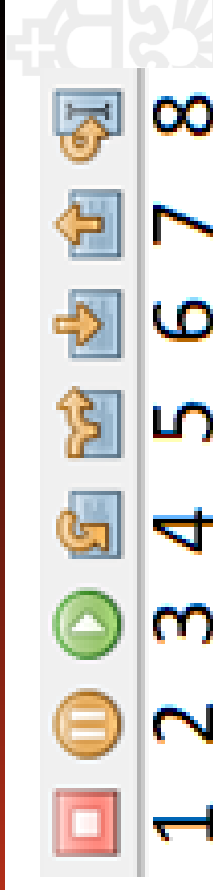
- Before the relevant lines of the bug
- Nearby the suspicious lines of code
- Where you would print the variables when not using the debugger

## The different commands I



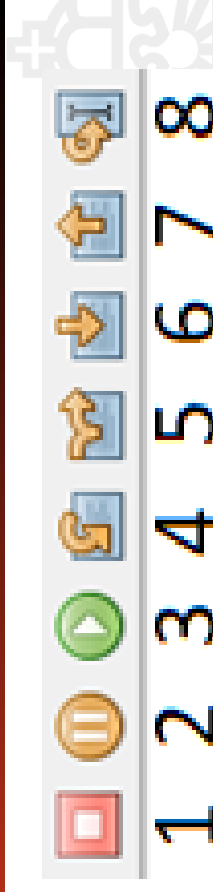
- 1 Finish Debugger Session (Shift + F5)
- 2 Pause
- 3 Continue (F5): Continue the program until the next breakpoint.
- 4 Step Over (F8): Execute the current line and break afterwards.

## The different commands II



- 5 **Step Over Expression (Shift + F8):** If the current line contains more functions, execute the current function and break before the next function in the expression.
- 6 **Step Into (F7):** If the current line contains a function, break the debugger at the beginning inside this function.

## The different commands III



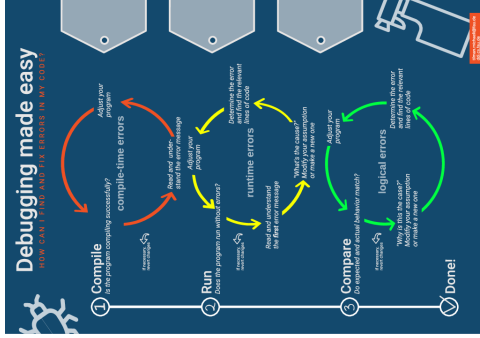
- 7 Step Out (Ctrl + F7): Finish the current function and break where this function was called.
- 8 Run to Cursor (F4): Same as Continue, unless the cursor is before the first breakpoint. In that case, it will break at the line of the cursor.

A demo about the different commands and when to apply them


Many other features to make debugging easier:

- Conditional breakpoints
- Exception breakpoints
- Watchpoints
- Modifying variables while running the debugger
- And many more...

- Introduction
- A systematic debugging procedure
- JavaDoc
- Evaluating a run time error
- The debugger
- Wrapping it up
- Time to practice



- Explained tools/concepts:
- JavaDoc (Compile, Run and Compare)
  - Evaluating a run time error (Run)
  - The debugger (Run and Compare)



**Radboud University Nijmegen**

Introduction  
A systematic debugging procedure  
JavaDoc  
Evaluating a run time error  
The debugger  
Wrapping it up  
Time to practice

Table of contents

Introduction

A systematic debugging procedure


JavaDoc

Evaluating a run time error

The debugger

Wrapping it up

Time to practice




**Radboud University Nijmegen**

Introduction  
A systematic debugging procedure  
JavaDoc  
Evaluating a run time error  
The debugger  
Wrapping it up  
Time to practice

Time to practice

- A complete debugging exercise can be found on Brightspace
- All the concepts occur
- Slides are available on Brightspace
- Any questions?




**Radboud University Nijmegen**

Introduction  
A systematic debugging procedure  
JavaDoc  
Evaluating a run time error  
The debugger  
Wrapping it up  
Time to practice

Thank you for listening!

Thank you for listening and I hope this was useful for you all!



**Radboud University Nijmegen**

Introduction  
A systematic debugging procedure  
JavaDoc  
Evaluating a run time error  
The debugger  
Wrapping it up  
Time to practice

References

Michaeli, T., & Romeike, R. (2019, October). Improving debugging skills in the classroom: The effects of teaching a systematic debugging process



## Useful Links

- <https://netbeans.org/kb/73/java/editor-codereference.html>
- [https://www.unomaha.edu/college-of-information-science-and-technology/computer-science-learning-center/\\_files/resources/CSLC-Helpdocs-HandlingJavaRuntimeErrors.pdf](https://www.unomaha.edu/college-of-information-science-and-technology/computer-science-learning-center/_files/resources/CSLC-Helpdocs-HandlingJavaRuntimeErrors.pdf)
- <https://www.jetbrains.com/help/idea/debugging-your-first-java-application.html>
- <https://www.fourkitchens.com/blog/article/step-step-through-debugging/>