

HOW DO EXPLICIT SPECIFICATIONS, A TEST-FIRST APPROACH, AND PROCEDURAL GUIDANCE TO CONVERT SPECIFICATIONS TO TESTS, HELP TO IMPROVE CODE QUALITY AMONG STUDENTS

‘SHALL WE DEFINE BLACK-BOX TESTS FOR THIS ASSIGNMENT? WE HAVE TO TEST IT ANYWAY’

by

Arno Broeders

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University, faculty of Science
Master Software Engineering
to be defended publicly on Day Month DD, YYYY at HH:00 PM.

Student number: 852104184
Course code: IM9906
Thesis committee: dr. ir. Sylvia (S.) Stuurman (chair), Open University
dr. ir. Harrie (H.J.M.) Passier (supervisor), Open University
prof. dr. Lex (A.) Bijlsma (supervisor), Open University

ACKNOWLEDGEMENTS

Graduating in a time of COVID-19 with lock-downs, no physical education and hardly any contact with my students involved in my research has cost me a lot of effort. Besides this graduation assignment I had many extra coordinating tasks that were necessary for the continuity of education of our Computer Science department at Avans Hogeschool Breda during these COVID-19 time. For me this time was characterized as stressful and exhausting where I often seeked for a spark of energy to continue working on my research.

Without a vast number of people who have continuously supported and motivated me during my research and writing of my thesis, I certainly would not have succeeded. While I owe my thanks to all of these people, there are a handful of people that I explicitly want to name.

First of all, I would like to express a very warm word of thanks to my thesis committee; chair dr. ir. Sylvia Stuurman and my supervisors dr. ir. Harrie Passier and prof. dr. Lex Bijlsma to let me graduate at their department. I would not have been able to complete this research without your support and motivational words. Thank you for all the tips, feedback, and your words of encouragement if my research was a bit disappointing on certain points.

I would like to thank my colleagues in the Computer Science program at Avans Hogeschool Breda for their interest in my research and their motivating words. Thank you for doing some of the tasks that I was unable to do during my Master study.

My sincere thanks to my direct colleagues in programming education who were directly involved in my research, ir. Ruud Hermans and ir. Dion Koeze. Thank you for your flexibility, for adopting the new teaching materials and for your time to argue and provide critical feedback on my research.

Last but not least, I am profoundly grateful to my wife Tew and our children Alex and Jasmine. Thank you for being there for me when we celebrated successes, but also for being there for me when I was sometimes frustrated or grumpy because things were not going the way I wanted. Often a loving look from you was enough to give me new energy, or to make me realize that I had to rest and move on another time. Thank you for your motivational words that finally got me across the finish line.

November, 2021
Arno Broeders

SUMMARY

As CS instructors, we want to deliver skilled professionals to the business world. Learning to program is often underestimated, not only by educators, but also by students themselves.

One of the programming-skills students must develop is quality awareness. One of the most important quality requirement is the correctness of the software that is developed. Students should ask themselves "When is the software I coded performing correctly?". This question is not so easy to answer because there is a discrepancy in the concept of correctness between the students and the professionals.

If we ask professionals: "When is software working correctly?", they will answer: "Software that works exactly according to specifications and is free of bugs.". Students are often satisfied when the output at a given value (roughly) matches what is expected and they tend to accept occasional bugs.

But what are those specifications? Often these are problem descriptions in natural language that must be solved by the software. Specifications in natural language are often not free from ambiguity and leave room for assumptions. That makes it difficult for novice programmers to code software and judging correctness.

In our research we will provide the students more formal specifications, leaving out room for ambiguity and assumptions. In these given specifications we may not only assume expected input, but unexpected input must also be specified for handling the right way. For documenting these specifications we will use the JML-standard.

In order for students to guarantee correctness of their code, we have to put them to testing. These semi-formal JML-specifications provide a structure that is easy to transfer to a set of automated tests. To help students transferring these JML-specifications to test-suits, we provide procedural guidance.

We will of course introduce these concepts extensively in updated teaching materials. And we test the result of this new methodology in a practical assignment.

The results from this practical assignment are promising. We see that adding semi-formal JML-specifications together with the training and procedural guide leads to more complete test sets and that the coded implementations score higher on correctness if test sets based on these specifications and procedural guide are made beforehand.

We also looked at the quality awareness of the students six months later. We see that the knowledge in the field of specifications and testing has remained, but that students no longer actively practice testing. We are somewhat disappointed in this, but our analysis, together with other academic research, gives us leads to investigate this further in the future.

CONTENTS

Acknowledgements	i
Summary	ii
1 Introduction	1
2 Background and Related Work	3
3 Research Questions and Methodology	7
4 Results	19
5 Conclusion	41
Bibliography	45
Appendix A: Procedural Guide	48
Appendix B: Lesson Material	50
Appendix C: The 'Milestone 6' assignment instructions	79

1

INTRODUCTION

Novice students in software engineering find solving even small problems in code extremely difficult. As CS instructors, we have to accept students have to go through some phases before understanding code quality and delivering the optimal quality in assessments. The first starting point of code quality for novice students is the correctness of code.

Novice students, starting to develop their coding skills often believe that their code is correct if it compiles. [Michaeli and Romeike \[2017\]](#) illustrates this clearly in their learning process view. They define five levels of comprehension in regards to software quality. The first novice level is when the compilation of code succeeds; students believe their code is correct. Where after a short time an output that looks plausible for a set of input values, without properly checking the results with the specifications, the software is already seen as working correctly. The achieved next step is when code has expected output based on some provided sample data, this was also found by [Edwards \[2004a\]](#). Concerning software quality, this will still not justify code correctness. The next three levels give the student the understanding that testing has a quality-enhancing effect, but testing is still difficult for students. [Bijlsma et al. \[2021\]](#) state students often try to get their code correct using a trial and error approach. We see this behavior in many students from observations in class when tests are provided together with an assignment. They believe the code will be correct if the tests pass at some point, not knowing if there are sufficient test-cases in the suit to prove full correctness. However, more importantly, this trial and error behavior appears to adversely affect the analysis and understanding of the code constructions [Bijlsma et al. \[2021\]](#). The ease with which they fiddle and try to get the tests passed seems to block analysis skills. [Edwards \[2004a\]](#) states that students must master basic comprehension and analysis skills; otherwise trial and error will become a handicap.

As instructors, we would like to see that students master comprehension and these analysis skills that result in awareness about code quality and not solely about passing tests. It makes sense to let students write their tests to gain these skills and awareness of code quality in this way.

The next hurdle is getting students to test. In our experience, students do not take tests of their own free will. Even if testing is mandated in assignments, it is often not performed. When rewarding tests with a bonus grade, tests are still not made often, or quickly rushed to get the bonus leading to uncomplete testsets. [Desai et al. \[2008a\]](#) adds that despite mandating testing, students continue to see it as an unnecessary burden. This is also a conclusion

by [Scatalon et al. \[2017b\]](#) who did a literature review on papers concerning software testing in an academic setting.

Although it is difficult to get students to test, we are firmly convinced that early learning of testing will positively influence software quality if students see its usefulness. With this research we want to see if we can alleviate the burden of testing by providing the students different tools. In addition, we want to pay explicit attention to external code quality, specifically the correctness of the code.

Tests come in 2 main variants [Myers et al. \[2012\]](#). White-box tests are tests that are written afterward to identify flaws in the implementation. By analyzing the implementation code all execution paths are followed to derive the test data. In many cases the test data will be derived at the neglect of the specifications so that any imperfections in the implementation will be taken over in the tests. Assessing correctness using these tests, therefore, seems less obvious.

Black-box tests are defined without looking at the implementation. They can be written before coding the implementation, but writing them after the implementation was made is possible too. The black-box tests usually form the specification for the implementation if they are written upfront the implementation and are well suited for a test-first approach. Well defined black-box tests will prove the correctness and completeness as part of the external code quality, of the implementation code. The challenge is, when is the set of tests sufficiently fully defined? If students write black-box tests based on the problem description, some cases are easily overseen; it results easily in happy-path testing leading to incomplete test-suites and thus incomplete specifications, as already shown in the [Edwards \[2004a\]](#) study. This way the expected increase in external code quality is not always met.

We believe that by making new students aware of external code quality, specifically the correctness of the code, students become more critical of their implementations. Black-box testing in the form of TDD supports this philosophy. We hypothesize that providing explicit informal specifications that cover the input variables of equivalency classes and all the boundaries should help students write well-defined test-suites without thinking about the implementation. By providing training on this from the start, directly after they master the first programming essentials, we believe that we give students new skills to grow into better software engineers.

Of course, we are not the first educators to plan quality improvements through testing in education. As mentioned before, we also have some hurdles to overcome. This makes it essential first to investigate what has already been done in this area, so we can add new insights to the box of knowledge. Chapter 2 discusses interesting findings that are useful for our research. Chapter 3 will formulate research questions and the methodology going to be used, to see whether we can add new results based on our methodology outside existing science. Chapter 4 then presents the process during the methodology of the research and the results. In chapter 5 we will form a conclusion and propose future work.

2

BACKGROUND AND RELATED WORK

The effects of testing on code quality among academic students have been studied by many researchers already. Most studies indicate that integrating tests in education has a quality-enhancing effect on the coding skills of the students [Desai et al. \[2008b\]](#).

Many novice students believe testing is done by compiling, running and validating the output manually against expectations [Michaeli and Romeike \[2017\]](#). The next logical step is to teach students how to use automated tests and that they are useful for regression testing. Automated tests consist of so-called test-suites with unit-tests covering a unit of work like a class or method. [Doorn \[2018\]](#) has found that only half of the universities in the Netherlands teach unit-testing in their curriculum, even though most teachers and students consider this an important topic.

Automated testing can be categorized into two forms, white-box and black-box tests [Myers et al. \[2012\]](#). White-box tests are tests that are based on the units already coded. Often they are created by following code execution paths and determining the test variables on the implementation logic. We see that this is not so effective as we would like. [Edwards and Shams \[2014\]](#) have shown that students with white-box tests achieve a considerable code coverage, but that most of the tests are happy-path testing. If test input variables are based on the already coded implementation it seems obvious that not all of the possible bugs are going to be found.

In contrast to White-box tests, black-box tests are not based on implemented code but the foreseen implementation specifications or problem description. [Leventhal et al. \[1994\]](#) discovered in their thinking-aloud study with students that students are positively biased when creating tests. Students make assumptions that can detract from correctness. If the problem domain is known, the assumptions can still yield some useful tests, but it does not guarantee a certain amount of correctness and completeness. In these cases many test-cases could be overseen.

2.1. COMPLEX LEARNING

Learning to solve real-world problems in computer algorithms and defining them in code is difficult for novice students in computer science. It requires programming language skills, abstracting problem situations, and creating algorithms that solve these problems under different variables. [Van Merriënboer and Krammer \[1987\]](#) describes various tactics of how instruction in programming could be most efficient for courses in high school.

For writing black-box tests this complexity is no different. Converting external specifications to black-box tests is a complex task. Next to conceptual knowledge, knowledge about coding, also knowledge about applying procedures is needed for students to apply such a procedural guide. [van Merriënboer and Kirschner \[2007\]](#) has provided 'Ten Steps To Complex Learning' to design educational materials for obtaining such complex skills. These ten steps help to design these educational materials following the four components that are needed to master complex skills:

1. **Learning tasks;** these are tasks that integrate skills, knowledge, and attitudes. They are organized from simple to complex, and support is diminished in each subsequent task.
2. **Supportive information;** is a guide that provides strategies for non-routinely learning tasks.
3. **Procedural information;** provides an algorithm for solving repetitive learning tasks.
4. **Task practise;** mastering the skills by repetitive exercises.

2.2. WHAT IS TEST-DRIVEN DEVELOPMENT?

Test-Driven Development or TDD is a principle that originated from the software development methodology extreme programming [Beck \[1999\]](#). It integrates black-box unit-testing tightly with writing implementation code. Before any code is written for the implementation, a suite of tests is created to cover the problem description. That is why it is called test-first. After writing the tests, the tests are seen as specifications for the implementation. Next, the implementation is written until all tests pass. A next optimization step is to refactor code so that the resulting code is readable and has the lowest possible complexity [Beck \[2003\]](#).

An important aspect of concentrating on testing first is that no implementation code has been created yet. Because no implementation exists when the tests are written, any thinking errors can arise during the implementation are not automatically taken over in the tests. Tests are based on equivalent class tests, assumed that the program algorithm has the same effect on a related set of inputs. To cover the exceptions to the equivalent classes, the series of inputs where the algorithm should start to behave differently are provided as boundary tests.

2.3. HOW DOES TDD HELP IN IMPROVING SOFTWARE QUALITY

If we are talking about software or code quality, we have to distinguish two different quality definitions.

- **External code quality.** These are various quality parameters that can be measured at the outside of the software unit. Think about performance of the software unit, but also the correctness of the code is an important aspect. Does the code do what we want, and are the exceptions to the happy-path correctly handled. If we are talking about unit-testing we have to explicitly specify the different cases, happy-path, and also including the exceptions to obtain a certain level of completeness in our test-suite. Only then can we guarantee the intended correctness of our implementation.
- **Internal code Quality.** This covers the maintainability of the code. How well is it structured, how complex is the code, is the code easy to read and maintain. Internal code quality is usually obtained by the refactoring step of the TDD methodology.

For our novice students we will find code correctness, of the external code quality, the most important aspect to start with.

[Bissi et al. \[2016\]](#) did literature study about the effectiveness of TDD. They collected 1107 articles between 1999 en 2014 about TDD and examined 27 of them in depth. They found that 88% of the studies identified an increase in code quality when applying TDD. If we concentrate on professionals and look at the [Bhat and Nagappan \[2006\]](#) study on the quality effects in industrial environments, we see that the effects on code quality are significant positive. Not all TDD studies are equally positive, [Scanniello et al. \[2016\]](#) found that TDD improves productivity at the cost of the internal quality due to the quick and dirty implementations to let the tests pass and not enough refactoring to increase the internal code quality. They found that a white-box test-last approach yields better internal quality than TDD.

2.4. TDD IN EDUCATION

A significant difference between professionals and novice students is experience. This difference is seen in the impact of TDD on code quality in general. Studies among professionals more often seem to measure a positive effect on code quality as studies in an academic setting as concluded by [Bissi et al. \[2016\]](#). From the eight studies of the quality impact with TDD in academics that [Bissi et al. \[2016\]](#) selected in his literature review, six studies concluded an increase in code quality, and one did not sense an increase but also did not negatively impact code quality.

The study 'Towards empirical evaluation of test-driven development in a university environment' of [Pancur et al. \[2003\]](#) compares TDD with a white-box testing variant and grades the improvements by TDD negative. This study was conducted with approximately 40 students. Half of the students used the white-box test-last approach and performed slightly better. One interesting result from this study is that students in the TDD group find TDD more helpful in requirements understanding and increasing code quality than students from the test-last group. We hope if we teach students a test-first approach in this study that students are more inclined to make testing their own because requirements will be better understood. Note that the results of this study by [Pancur et al. \[2003\]](#) were preliminary because of the overlap in time with the publishing of the article and analyzing the

experiment results. Therefore we find this article not entirely valid. After reading this article, we also find it unclear if only external code quality, specifically correctness, is measured or if there is a mix-up with internal code quality.

When teaching novice students to code we find external code quality, namely the correctness aspect of code more important than internal code quality. In more advanced courses we will focus more on internal quality and maintainability of code. The use of a test-first approach seems justified because of this reason in novice courses.

[Polikarpova et al. \[2013\]](#) takes it one step further and uses strong formal specifications for the problem analysis and convert them to black-box tests automatically. They found that these black-box tests based on strong specifications double the amount of found bugs at reasonable effort.

Formal specification will be too academic for Universities of Applied Science and also some properties will be very hard to describe formally, so we cannot use this automatic conversion. Not using automatic conversion is not a problem, we believe that manually converting specifications could trigger more awareness of common pitfalls in test cases. If we provide semi-formal specifications in the form of JML (Java Modelling Language) and specify the pre- and postconditions in java pseudo-code we think we give the students a useful tool that allows them to easily develop tests with less complexity involved.

3

RESEARCH QUESTIONS AND METHODOLOGY

To help novice students in their journey to become good software engineers, we, as CS educators, want students to write correct code. In addition to the correctness of code, we want students to be aware of this software quality aspect and critically and analytically look at their code.

In this research we want to limit our scope of software quality to correctness aspect of the external code quality. We think the other code quality parameters are more suitable for the more advanced students.

We believe that a test-first approach helps students increase their awareness of software quality, and we also believe that providing explicit stated informal specifications helps students increase their analytical skills by reducing the complexity of transferring a problem description to implementation logic at this stage. If the provided specifications are complete, we can provide students with a procedural guide to convert these specifications into a complete suite of black-box tests. Students have to follow the guide step-by-step to transfer the specifications to a fully defined test suite without too much complexity so they can focus on input variables derived from the specifications and learn to automate the writing tests process. If this results in correct and complete test suites, the external code quality will also increase according to our hypothesis. We also hope that this way students will be more willing to write tests with little effort so that they will produce higher quality code.

Although a test-first approach is automatically a black-box test and TDD focuses on defining tests before the implementation, we will explicitly use the term test-first in this research. We use the term test-first because black-box tests can also be written after the implementation was made, and we do not use the term TDD because TDD is more than simply defining and writing tests upfront. Even better would be the term test-first specification driven testing, because tests are going to be designed based on the provided JML-specifications. For simplicity reasons we just use test-first.

The big difference with other research is that we investigate the effects of provided explicit informal specifications and provide students a procedural guide to transforming these specifications into black-box tests. Therefore we will answer the following research question to validate our hypothesis:

How do explicit specifications, a test-first approach, and procedural guidance to convert specifications to tests, help to improve code correctness among students?

To conduct this research, we are expanding the current 'Programming 2' course with two lessons on software quality and testing. Each lesson consists of two hours of theory and three hours of supervised practice in small study groups of approximately six students. A total of ten hours of education is therefore invested in software quality and testing in this novice course.

The existing group assignment 'CodeCademy' that is already in this course to assess programming skills, will be expanded with an explicit test component. To prevent students from showing procrastination and to provide them with feedback early, there is an option for students to follow a set time path in milestones in which they each deliver a partial product and receive early feedback on this. We have packaged the test assignment that is used in the context of this research in 'Milestone 6'.

The ultimate goal of these two lessons and the 'Milestone 6' assignment would be that students become quality conscious and deliver high quality code. Because of this, we will not only examine the products of these assignments but also the behavior of the students during and after the 'Milestone 6' assignment. To analyze the behavior of students during the 'Milestone 6' assignment we will ask them to record the development environment and verbal communication and send these in as part of the assignment. This recording will be referred to as the 'thinking-aloud' assignment. To analyze the behavior afterwards, we will conduct interviews and examine if students will write automated tests by themselves in other assignments or projects. To examine all these elements and structure our research we will break-down this question in the following sub-questions:

RQ1: What does a procedural guide to transfer external specifications to black-box tests look like and are there minimum requirements for the external specifications.

[Polikarpova et al. \[2013\]](#) uses formal specifications that are to be automatically converted to suites of black-box tests. It means that there is an algorithm that does so. Our target group are first-year students of a University of Applied Science for which formal specifications could be too academic. In addition to the academic character, it is very difficult to formally describe certain conditions or states in software. Using more informal but complete specifications prohibits automatic conversion to black-box tests, but students' interpretation, together with a procedural guide should be enough to let the students write their well-defined suite of black-box tests. We will design such a step-by-step procedural guide that students can use to write such a suite of black-box tests that are based on less formal external specifications. To help students even more, we will partition the input variables into equivalency classes so students only have to think about inputs variable in the range and at the boundaries of these equivalency classes. By providing also subcontracts for illegal preconditions we omit the fact that students tend to write only happy-path tests and make them aware that also invalid preconditions need tests for adequate exception handling.

Method: First of all we examined the JML format for the specifications. This JML format allows us to write down explicitly the pre- and postconditions and also any exceptions that may occur at a specific precondition. To remove unnecessary complexity at this stage of the

course we provide the partitioned equivalency classes of these specifications so our novice students do not have to think about partitioning themselves. We investigated step-by-step what would be needed to setup a well-designed test suite. Therefore we made several test-suites upfront by our self (the teacher-set for the 'Milestone 6' assignment) and identified the following five main concepts:

- Prepare the testsuite/project.
- Design tests for each subcontract
- Code the tests
- Code the implementation
- Test the implementation

Each of these main concepts consists of small steps to complete each of these concepts. These small steps are written down, categorized by concept and numbered so that students can go through all the steps one-by-one and repeat the step-by-step guide until the complete specifications of the method have been converted into a complete test set. To make this procedural guide even more useful for students, a glossary of terms from the test lessons has been added to the procedural guide. For the completed guide see Appendix A.

RQ2: How do we teach students to apply the procedural guide to create a complete test-first-suite from external specifications.

If we want to activate students to develop through a test-first approach, then we must ensure that students see the importance and benefits of this methodology. We need to teach them what external code quality means in terms of correctness and completeness to create support for this methodology. Novice students find it difficult to define black-box tests without thinking about the implementation details, especially when only a problem description is given. Therefore we teach them step-by-step on how to convert semi-formal external specifications into black-box tests using the procedural guide. We develop the instructions inspired by the four component instructional of [van Merriënboer and Kirschner \[2007\]](#). We supply supportive information and we practise step-by-step the assignments, where with each practice iteration we drop a piece of given information so that the students learn to automate the definition and writing of tests in a natural way. We hypothesize that if the process is easy enough, students stop thinking about implementation when converting the external specifications into the black-box test-code. Analyzing the exercises' test-code and analyzing the 'thinking-aloud' recordings afterward will give insight into how much this skill to transfer the external specifications into black-box tests routinely is matured.

Method: Because we have two lessons in two consecutive weeks, we split the lectures of theoretical content into two distinct subjects and let the students practise directly after the lectures. Lesson one will cover the concept of code quality, correctness and the design of test cases, while lesson two will focus on the use of JUnit to automate the test-cases.

Lesson one starts with 'the need for software quality'. We will ask the students to analyse a piece of black-box software for correctness without giving the full specifications for the software. The only specification students have in hand is the name of the software-unit 'calculateAge' and the parameter that is required in the form of `<dd-mm-yyyy>`. The software-unit contains intentional mistakes, but behaves correctly in certain situations.

The code in Listing 3.1 was compiled to a class file and students are asked to analyse this software-unit by executing '*java calculateAge <dd-mm-yyyy>*' from the commandline, not knowing any implementation details.

```
import java.time.LocalDate;
import java.time.Month;
import java.time.format.DateTimeFormatter;

public class calculateAge {
    public static void main(String[] args) {
        LocalDate date = LocalDate.parse(args[0],
            DateTimeFormatter.ofPattern("d-M-yyyy"));
        int age = LocalDate.now().getYear() - date.getYear();
        System.out.println("Als je geboren bent op " +
            date.format(DateTimeFormatter.ofPattern("d-M-yyyy")) + " ben je "
            + age);
    }
}
```

Listing 3.1: *calculateAge* method used for black-box testing

We predict that a significant amount of students will state that the provided black-box software-unit is correct as students tend to check this assignment only with their own date of birth. As seen in the source-code, there is an intentional mistake to only subtract the dates year components to calculate the age. This only leads to correct results if their birthday has already passed in the current year.

The outcomes of this exercise gives a clear view on why testing multiple partitions of equivalency classes is important. It also provides a starting point for teaching the essence of clear specifications and not to make assumptions. We will confront students with the question why they assume the age in years when the unit should actually return the age in months, making the implementation even more incorrect. Without specifications we can only guess the intended operation.

Testcase	Not born yet
Preconditie	<u>dateOfBirth = LocalDate.now().plusDays(10)</u>
Orakel	<u>IllegalArgumentException</u>
Postconditie of Exception	<u>IllegalArgumentException</u>

Figure 3.1: Test template

After this first part we are going to deal with specifications in JML-format, pre-conditions, post-conditions, exceptions and oracles. We use the specifications of the software-unit

'*daysUntilNextBirthday*' as example and cover step by step the different subcontracts of the specifications and the intended behaviors of this method. We omit the leap year for complexity reasons, but provide it as extra exercise to students looking for extra challenge.

We give the specification in Listing 3.2 to the students and we will design sufficient test-cases for each subcontract together during the lesson. We write down these test-cases on paper without actually applying them at this stage. To help students in designing test-cases. A stripped-down version of the procedural guide is provided with only the steps to design the test-case and outcome of the oracle. For the design of the tests the template in Figure 3.1 is provided to the students:

```
/**
 * @desc Calculates the amount of days until the next birthday of dateOfBirth.
 *
 * @subcontract: not born yet {
 *   @requires dateOfBirth > LocalDate.now();
 *   @signals (IllegalArgumentException) dateOfBirth > LocalDate.now();
 * }
 *
 * @subcontract: null dateOfBirth {
 *   @requires dateOfBirth == null;
 *   @signals (NullPointerException) dateOfBirth == null;
 * }
 *
 * @subcontract: birthday is yet to come this year {
 *   @requires LocalDate.now().getMonthValue() < dateOfBirth.getMonthValue()
 *   ||
 *   LocalDate.now().getMonthValue() == dateOfBirth.getMonthValue() &&
 *   LocalDate.now().getDayOfMonth() < dateOfBirth.getDayOfMonth();
 *   @ensures \result = #days until (not including) next birthday &&
 *   1 <= \result < 365;
 * }
 *
 * @subcontract: today is my birthday {
 *   @requires dateOfBirth.getMonthValue() == LocalDate.now().getMonthValue()
 *   && dateOfBirth.getDayOfMonth() == LocalDate.now().getDayOfMonth();
 *   @ensures \result = 365;
 * }
 *
 * @subcontract: birthday has already passed this year {
 *   @requires dateOfBirth.getMonthValue() > LocalDate.now().getMonthValue() ||
 *   dateOfBirth.getMonthValue() == LocalDate.now().getMonthValue() &&
 *   dateOfBirth.getDayOfMonth() > LocalDate.now().getDayOfMonth();
 *   @ensures \result = #days until (not including) next birthday &&
 *   1 <= \result < 365;
 * }
 */
public static int daysUntilNextBirthday(LocalDate dateOfBirth);
```

Listing 3.2: JML-specifications for *daysUntilNextBirthday*

After defining all the different testcases based on the specification we will provide the compiled class-file from Listing 3.3 to students to let them analyse in command line the correctness based on the prepared test cases.

```
import java.time.LocalDate;
import java.time.Period;
import java.time.format.DateTimeFormatter;

public class daysUntilNextBirthday {

    public static void main(String[] args) throws Exception {
        LocalDate date = LocalDate.parse(args[0],
            DateTimeFormatter.ofPattern("d-M-yyyy"));

        if(date.isAfter(LocalDate.now())) {
            throw new IllegalArgumentException("dateOfBirth cannot be in the
                future");
        }

        int days = 0 - Period.between(LocalDate.of(LocalDate.now().getYear(),
            date.getMonth(), date.getDayOfMonth()), LocalDate.now()).getDays();
        if(days<0) {
            days += 365;
        }
        System.out.println("It takes " + days + " days until your next
            birthday (born on: " + args[0] + ")");
    }
}
```

Listing 3.3: *daysUntilNextBirthday* method used for black-box testing

The powerpoint slides of the lecture are show in Appendix B. After class there is a three hour guided workshop where students practice preparing tests, fill them in on 'paper' in a given template and afterwards test a black-box compiled unit of software via the command line.

Following assignments (Listings 3.4, 3.5) are given to the students, these JML specifications are provided to the students along with a set of compiled class-files (Listings 3.6, 3.7), so students can apply their designed tests on the implementation themselves. The implementations are hidden and they all contain some intentional bugs for the students to find.

```
/**
 * @desc A company orders an application that needs to calculate the annual
 * bonus of its employees. This bonus is a percentage of their monthly
 * salary, and depends on how long they have worked for the company.
 *
 * @subcontract: negative years in service {
 * @requires yearsInService < 0;
 * @signals (IllegalArgumentException) yearsInService < 0;
 * }
```

```

*
* @subcontract: less than three years at the company yields a bonus of 0% {
*   @requires 0 <= yearsInService <= 3;
*   @ensures \result = 0;
* }
*
* @subcontract: more than three years at the company yields a bonus of 50% {
*   @requires 3 < yearsInService <= 5;
*   @ensures \result = 50;
* }
*
* @subcontract: more than five years yields a bonus of 75% {
*   @requires 5 < yearsInService <= 8;
*   @ensures \result = 75;
* }
*
* @subcontract: more than eight years yields a bonus of 100% {
*   @requires 8 < yearsInService;
*   @ensures \result = 100;
* }
*/
public static int getAnnualBonusPercentage(int yearsInService);

```

Listing 3.4: JML-specifications for *getAnnualBonusPercentage*

```

/**
* @desc A hardware store sells hammers (5 euros) and screwdrivers (10 euros).
* This method can calculate the price a customer needs to pay when buying
* these products.
*
* @subcontract: invalid input negative totalDue {
*   @requires totalDue < 0.0;
*   @signals (IllegalArgumentException) totalDue < 0.0;
* }
*
* @subcontract: invalid input negative amount of screwdrivers {
*   @requires amountOfScrewdrivers < 0;
*   @signals (IllegalArgumentException) amountOfScrewdrivers < 0;
* }
*
* @subcontract: If the total is less than 200 euros, no discount {
*   @requires 0.0 < totalDue <= 200.0;
*   @ensures \result = totalDue;
* }
*
* @subcontract: If the total is more than 200 euros and buys no more than 30
* screwdrivers, then the client obtains a discount of 5% over the total {
*   @requires 200.0 < totalDue <= 1000.0 && 0 <= amountOfScrewdrivers <=30;
*   @ensures \result = totalDue * 0.95;

```

```

* }
*
* @subcontract: If the total is more than 200 euros and buys more than 30
* screwdrivers, then the client obtains a discount of 15% over the total {
* @requires 200.0 < totalDue <= 1000.0 && 30 < amountOfScrewdrivers;
* @ensures \result = totalDue * 0.85;
* }
*
* @subcontract: If the total is more than 1000 euros and buys no more than
* 30 screwdrivers, then the client obtains a discount of 20% over the
* total {
* @requires 1000.0 < totalDue && 0 <= amountOfScrewdrivers <=30;
* @ensures \result = totalDue * 0.80;
* }
*
* @subcontract: If the total is more than 1000 euros and buys more than 30
* screwdrivers, then the client obtains a discount of 30% over the total {
* @requires 1000.0 < totalDue && 30 < amountOfScrewdrivers;
* @ensures \result = totalDue * 0.70;
* }
*
*/
public static double getDiscountedPrice(double totalDue, int
    amountOfScrewdrivers);

```

Listing 3.5: JML-specifications for *getDiscountedPrice*

```

public class getAnnualBonusPercentage {
    public static void main(String[] args) throws IllegalArgumentException {
        int input = Integer.valueOf(args[0]);
        if(input < 0 ) {
            throw new IllegalArgumentException("yearsInService must be
                positive");
        } else if (input < 3) { //bug, moet <= zijn
            System.out.println("0");
            return;
        } else if (input < 5) { //bug, moet <= zijn
            System.out.println("50");
            return;
        } else if (input < 8) { //bug, moet <= zijn
            System.out.println("75");
            return;
        } else {
            System.out.println("100");
        }
    }
}

```

Listing 3.6: *getAnnualBonusPercentage* method used for black-box testing

```
public class getDiscountedPrice {
    public static void main(String[] args) throws Exception {
        if(args.length==0) {
            return;
        }

        double totalDue = Double.valueOf(args[0]);
        int amountOfScrewdrivers = Integer.valueOf(args[1]);

        if(totalDue < 0.0) {
            throw new IllegalArgumentException("totalDue must be positive");
        }

        if(amountOfScrewdrivers <= 0) {
            throw new IllegalArgumentException("amountOfScrewdrivers must be
                positive");
        }

        if(totalDue <=200.0 && amountOfScrewdrivers > 30) { //bug
            System.out.println(totalDue * 0.9);
            return;
        }

        if(totalDue <=200.0) {
            System.out.println(totalDue);
            return;
        }

        if(totalDue > 200 && totalDue <=1000.0 && amountOfScrewdrivers <= 30)
            {
                System.out.println(totalDue * 0.95);
                return;
            }

        if(totalDue > 200 && totalDue <=1000.0 && amountOfScrewdrivers > 30) {
            System.out.println(totalDue * 0.85);
            return;
        }

        if(totalDue > 1000 && amountOfScrewdrivers < 30) { //bug
            System.out.println(totalDue * 0.80);
            return;
        }

        if(totalDue > 1000 && amountOfScrewdrivers >= 30) { //bug
            System.out.println(totalDue * 0.75);
            return;
        }
    }
}
```

```
        System.out.println(totalDue);
    }
}
```

Listing 3.7: *getDiscountedPrice* method used for black-box testing

For lesson two, a week after, we will dive a bit deeper in the theoretical aspects of black- and white-box testing, the test-first approach and we concentrate on the JUnit framework and automating tests. After the basics in JUnit testing, we will learn the students to apply the steps of the procedural guide to transform the specifications to a set of unittests. This applying of the procedural guide is based on a case-method *getWordRating* with the specs show in Listing 3.8. We will write a complete test-suite together with the students, using a test-first design approach. The first subcontract is given as demonstration. For the next subcontracts students are given a little more freedom until they can independently convert the last subcontract into a set of unit tests.

```
/**
 * @desc returns a dutch word-rated grade based on a numeric grade.
 *
 * @subcontract out of range grade {
 * @requires grade < 1 || grade > 10;
 * @signals (IllegalArgumentException) grade < 1 || grade > 10; }
 *
 * @subcontract inadequate {
 * @requires 1 <= grade < 5,5;
 * @ensures \result = "onvoldoende"; }
 *
 * @subcontract adequate {
 * @requires 5,5 <= grade < 7;
 * @ensures \result = "voldoende"; }
 *
 * @subcontract ample {
 * @requires 7 <= grade < 8;
 * @ensures \result = "ruim voldoende"; }
 *
 * @subcontract good {
 * @requires 8 <= grade < 9;
 * @ensures \result = "goed"; }
 *
 * @subcontract excellent {
 * @requires 9 <= grade <= 10;
 * @ensures \result = "uitmuntend"; }
 */
public static String getWordRating(double grade);
```

Listing 3.8: JML-specifications for *getWordRating*

At the end of the second lesson the 'Milestone 6' assignment will be explained, an explanation of the methods to be implemented and an explanation of what is expected of the students in the 'thinking-aloud' assignment. Because this is the last lesson before the

Christmas holidays, we give students three weeks to complete the 'Milestone 6' assignment. The instructions of the 'Milestone 6' assignment are also provided in detail as pdf document (see Appendix C). Students are motivated to start this assignment immediately after the lecture, but we expect that many students will drop out and enter holiday mode as this is the last day of college this year.

RQ3: What difference do we observe in coding behavior between students who use a test-first approach based on external specifications together with the procedural guidance, and students who do not use testing but have access to the specifications.

We want to know if a test-first approach will improve code correctness beyond simply using the semi-formal external specifications directly for the implementation. This research question focuses on the behaviour of students during the 'Milestone 6' assignment. In this research question we do not look into the technical analysis of the completeness of the test-set and correctness of the implementation, we will do that in the next research question. We hypothesize that students will make test-suites for the implementations where is test-set is optional if they understand the benefits and they make the tests with ease. We hope to find signs that students themselves steer towards a test-first approach.

Method: For the mandatory group-assignment 'CodeCademy' for the programming 2 course, student groups of four students were composed. We will motivate students to receive an extra bonus of 1 point to participate in the 'Milestone 6' assignment, which is a partial product of the larger group assignment. To participate in the 'Milestone 6' assignment, students must work in pairs, so there are two pairs for each 'CodeCademy' group. For the 'Milestone 6' assignment four methods are prescribed complete with specifications. The two pairs agree among themselves for which of the two methods they each will develop a mandatory test-set so that the two pairs deliver a complete test set for all four methods. These four test sets are required in the final group assignment which is mandatory to pass the course. In addition to the two mandatory test sets per couple, each couple makes the four implementations. Two implementations are made using a test-first approach of the two developed testsets and for the two other implementations students are allowed to make implementations according to the specifications at their own discretion. Making additional test-sets is therefore allowed and we can deduce from this whether students are willing to test these two implementations with test sets as well. During the assignment all coding and verbal communication is recorded and handed in as part of the 'Milestone 6' assignment. We will analyze their way of working and look if we can observe some changes in behavior and quality awareness. We may decide not to view all recordings if it appears that we have sufficient observations to describe the behavior and to provide a substantiated answer to this research question.

RQ4: What improvement do we see in the code correctness when using a test-first approach together with explicit informal specifications and the procedural guide. Despite the fact that many studies already show that testing can certainly improve code-quality, we want to investigate whether this is still the case when explicit specifications are given in JML. It could well be that the specifications are so explicit that the specifications already ensure an error-free implementation. As a result, it may well be that making tests in ad-

vance is a waste of time because no increase in correctness is measurable. In addition to the implementation code correctness, we also want to measure the completeness of the test-suites. Historically, we see that students quickly slide into happy path testing. We expect that due to the explicit partitioning of the preconditions and the procedural guide, students will certainly also test the exceptions in addition to the happy-paths.

Method: Students' submissions of the 'Milestone 6' assignment, with and without test-suites can be checked on correctness on a teacher test set that is fully reviewed. It is important to know whether a test-first approach in concern to correctness is of added value in addition to only providing explicit specifications so we can accept our hypothesis. In addition to analyzing the correctness of the implementation, we also want to assess the correctness and completeness of the created test sets. To do this, we will apply two methods:

- To check the correctness of the test sets, we will test each student submitted test set against the correct teacher-implementation.
- In order to view the completeness of the test sets, we will modify the correct teacher-implementations that lend themselves to this by means of mutation testing. Any mutation of the teacher implementation must lead to a failure of at least one test. If this does not happen, the test set is not complete.

RQ5: How do students feel helped by using a test-first approach together with the procedural guidance. It is important for the willingness to test that students find the tests useful. We want to know what helps students the most and if they are convinced that following a test-first approach with the procedural guide of converting external specifications leads to better external quality of code.

Method: After analysing the 'thinking-aloud' assignment we will define a list of subjects and striking things that we noticed that will be used for our semi-structured interview. Also we want to focus on how the procedural guide and external specifications in JML format has helped the students. We will conduct these interviews after a few months when students are working on a free project so students can reflect on their testing skills in later assignments and projects. This gives us the opportunity to immediately monitor how students deal with tests a while later and to analyse how much knowledge about testing has remained during a number of months.

4

RESULTS

In order to answer the different research questions, we analyzed and combined information obtained from different parts of the research.

- To create test cases, we need input variables and intended results or intended exceptions for each partition. We studied the JML standard and looked at how to capture this information as clearly as possible. JML already provides parameters for capturing the necessary information, however, capturing the multiple partitions is less clearly defined in the JML-standard than we would like. The QPED research group has found an alternative notation using subcontracts that distinguishes the different partitions much more clearly than the @also keyword in JML. The alternative notation used is the @subcontract keyword with a descriptive text and the pre- and post-conditions (or signals) between braces behind this, see the example in Listing 4.1.

```
/**
 * @subcontract <description> {
 *   @requires <precondition>
 *   @ensures <postcondition> }
 **/
```

Listing 4.1: The use of @subcontract in JML-specifications

- We designed lessons inspired by the four component instructional design of [van Merriënboer and Kirschner \[2007\]](#). We conducted these lessons and looked closely at how students picked up the lesson material during the practical. Specific attention was paid to the number of questions during class and after that, what these questions were related to and what the students' working methods were. A few months later we asked students questions about the lessons in the interview and the questionnaire.
- We have analyzed the 'Milestone 6' assignment. About 50% of the students participated (52 out of 102 students) in this assignment. This 'Milestone 6' assignment consists of three products for each student-pair: The four implementations based on the specifications, at least two test suites based on the specifications and a video recording of the 'thinking-aloud' session while making the test-suites and implementations. For the 'thinking-aloud' analysis, we only analysed 11 out of 25 recordings in

depth. After these 11 recordings we concluded that no new information was discovered. The other 14 recordings we scanned quickly but we could not discover any surprising facts.

- Five months after the lectures about code-quality and testing we held interviews to analyse the testing-behavior at that point. Because only six students were willing to participate in the interviews, a questionnaire was drawn up based on the most striking facts from these interviews. By raffling a prize among the entries, twenty students took part in the survey so we can create a more solid basis on which to base conclusions.

4.1. PROCEDURAL GUIDE

RQ1: What does a procedural guide to transfer external specifications to black-box tests look like and are there minimum requirements for the external specifications.

We want to design a procedural guide as concise as possible, there should be no reading barrier to give support to students in the process of converting the specifications into test suites. If a student no longer knows how to define tests, he should be able to understand this again with as little text as possible. Meanwhile, we also want it to be helpful in understanding the technical terms we use. We will design the guide with these aspects in mind and provide the step-by-step instructions necessary to convert JML specifications to test-suites along with a glossary of technical terms and keywords that are used while designing and writing tests.

To design a test-suite, we need to know the different input-domain partitions. Each value specified in the input-domain of a single partition will have a similar effect on the post-condition will be an equivalence class or subcontract. These different partitions will form the collection of subcontracts for the complete specification of the software unit¹. For each subcontract the range of valid input values is called the pre-condition. If there is a valid post-condition for this given pre-condition we can define an oracle for this and compare it to the return value of the implementation under test. If the input-domain of the subcontract leads to an invalid post-condition there will be an exception specified. So the only required parameters to base the test-suites on are the pre-condition and post-condition, or a pre-condition and exception for each subcontract. Special attention must be paid to the fringe cases for which this subcontract is just applicable. These are the so-called boundaries of the equivalence classes and must certainly be tested for correct functioning. For clarification of the subcontract there is also a less formal description added. This description only serves as a label for the subcontract and no pre- or post-condition needs to be derived from this.

The main phases of the procedural guide are: Prepare the testsuite, design the tests, code the tests, code the implementation and test the implementation. To guide the students through these phases we identified all the small steps to take to complete each phase by making the teacher reference test-sets. The result of the procedural guide can be seen in

¹For simplicity reasons we will assume there is a one-to-one relationship between subcontracts and partitions. When defining subcontracts in the specifications we cannot always foresee all the different partitions in a more complex software-unit. This could result in an iterative process where subcontracts are later expanded when needed. For this research and the assignments for the students we can assume this one-to-one relationship.

4.2. INSTRUCTION

RQ2: How do we teach students to apply the procedural guide to create a complete test-first-suite from external specifications.

We have provided two theory lectures of two hours each followed by a supervised practical of three hours.

In lesson one, we made students aware of the concepts of quality and specifically correctness. The exercise to test the software unit *calculateAge()* without specification turned out well. There was a clear divide among the students. Students who thought the implementation was good and students who saw that the implementation was wrong. This provoked a good discussion after which, after demonstrating which data leads to wrong output, it soon became clear what went wrong in the implementation. Students were again surprised when we told them that the output should actually be the age in full months instead of years. It quickly became clear that without specifications we cannot prove whether a software unit is correct.

We showed students that specifications in descriptive text are often ambiguous. An example is "choose a number between one and ten". Are the numbers one and ten included? We would say not by the word 'between', but a certain context might make us decide to include one and ten as a valid choice. Of course, we do not want to develop software based on assumptions, we want a clear and unambiguous specification so we showed students a more formal notation in JML.

We explained students how the specifications according to the JML standard can be properly and clearly read. Concepts such as pre-, post-condition and exception and their JML equivalents @requires, @ensures and @signals are explained and demonstrated. We have explicitly made it clear to students that for now the intention is not to be able to design JML-specifications at this point, but only to be able to read and understand them.

The next step in the lesson was to clarify equivalent classes. For the rest of the lesson, the example *daysUntilNextBirthday()* was used, the specifications of this unit are found in Listing 4.2. But those specifications are not immediately shown to the students.

```
/**
 * @desc Calculates the amount of days until the next birthday of
 * dateOfBirth.
 *
 * @subcontract: not born yet {
 *   @requires dateOfBirth > LocalDate.now();
 *   @signals (IllegalArgumentException) dateOfBirth > LocalDate.now();
 * }
 *
 * @subcontract: null dateOfBirth {
 *   @requires dateOfBirth == null;
 *   @signals (NullPointerException) dateOfBirth == null;
 * }
 *
 * @subcontract: birthday is yet to come this year {
 *   @requires LocalDate.now().getMonthValue() <
```

```

*     dateOfBirth.getMonthValue() ||
*     LocalDate.now().getMonthValue() == dateOfBirth.getMonthValue() &&
*     LocalDate.now().getDayOfMonth() < dateOfBirth.getDayOfMonth();
* @ensures \result = #days until (not including) next birthday &&
*     1 <= \result < 365;
* }
*
* @subcontract: today is my birthday {
* @requires dateOfBirth.getMonthValue() ==
*     LocalDate.now().getMonthValue() &&
*     dateOfBirth.getDayOfMonth() == LocalDate.now().getDayOfMonth();
* @ensures \result = 365;
* }
*
* @subcontract: birthday has already passed this year {
* @requires dateOfBirth.getMonthValue() >
*     LocalDate.now().getMonthValue() ||
*     dateOfBirth.getMonthValue() == LocalDate.now().getMonthValue() &&
*     dateOfBirth.getDayOfMonth() > LocalDate.now().getDayOfMonth();
* @ensures \result = #days until (not including) next birthday &&
*     1 <= \result < 365;
* }
*/
public static int daysUntilNextBirthday(LocalDate dateOfBirth) throws
    Exception;

```

Listing 4.2: JML-specifications for *daysUntilNextBirthday()*

Students were asked to distinguish different input values that lead to a different code-path to calculate the number of days until the next birthday. We try to give students a sense of what the different partitions or subcontracts are used for. This proves to be extremely difficult for students. After giving them a few minutes to think, we gave them a few suggestions such as: What if the date of birth is an invalid date, for example January 32; What if the date of birth is in the future, and is that actually possible? Should a choice like this be stated in the specifications?

Students are still unable to come up with partitions. The suggestion: 'What if the date of birth is an invalid date', started a nice discussion. A considerable group of students commented that an invalid date cannot exist in a date-type variable. For us, this showed that students are following the lesson well and are seriously considering our options.

After a while, we decided to continue with hints and follow-up conversations. Our next hint was, how many days will it be until your next birthday if your birthday is today. We suggested the value 0 which was blindly accepted by the students. After this, we emphasized 'until your next birthday'. Some students responded promptly with 365 days. The follow up question by us was, but what if a leap year is involved? It quickly becomes apparent to students that distinguishing different partitions is a difficult task. We tell them that we will indicate the different partitions in the beginning, but that they will eventually have to take a good look at them themselves. At this stage we introduced the concept subcontract and we showed how to capture such a specific situation in a subcontract. Support in this way has also been suggested by the four component model of [van Merriënboer and Kirschner](#)

[2007]. Start with a demonstration, then take small steps, each more challenging in each iteration.

We then proceed to deal with the next subcontract: 'The birthday has yet to come this year'. We introduce the concept of a testing-oracle and teach students to choose a value from the input domain that produces a result that is always the same. Students all come with a fixed date, often if their own birthday is still happening this year, with their own date of birth. They do not yet see that such a test only works temporarily and that the expected outcome is different at another time. We will go into this in more detail.

As an example, it is mentioned that a fixed birthday on January 2nd is a bad choice. Tomorrow the result of the number of days until my next birthday has changed again. So the oracle leads to a non constant expected test result. By using such variable test results, we would need the implementation to calculate the intended test result, which of course renders the test useless. Students find it difficult to find an input variable that is relative to today. After the suggestion 'three days after today' it suddenly became clear to the students what is meant by a date relative to today.

For all subcontracts we defined test cases together with the students and recorded them in a given template. The testcase refers to the subcontract that is being tested, the precondition is a valid value in the input domain, oracle is a constant test output reasoned with the test oracle. The post-condition or exception is the actual output of the implementation under test. If the output of the oracle and the post-condition are equal, the test is passed. The used template is shown in figure 4.1.

Testcase	Not born yet
Preconditie	<code>dateOfBirth = LocalDate.now().plusDays(10)</code>
Orakel	<code>IllegalArgumentException</code>
Postconditie of Exception	<code>IllegalArgumentException</code>

Figure 4.1: Test template

After this example with complete analysis of pre- and post-condition and designing the testcases with also the edge-cases covered we showed students a procedure on how to setup test-cases for a subcontract. This procedure is a small subset of the complete procedural guide and only covers the design of the testcases for each subcontract and to validate the implementation on correct behaviour. These steps are taught to students as follows:

1. Make **at least** 1 test case for each @subcontract (think of edge cases, 1 test is seldom enough).
2. For each test case, come up with a precondition that satisfies @ensure and leads the oracle that leads to a constant output value.

3. Run the method under test with the precondition from step 2.
4. Verify that the post-condition matches the output from the oracle from step 2.
5. If step 4 is positive, the test is passed.

The theory lesson is concluded with a discussion of this small part of the procedural guide that deals with converting specifications into test cases.

The theory lesson is followed by a three-hour practical, supervised by teachers. There are three assignments for the students to complete. The first assignment is only testing of the *daysUntilNextBirthday()*. The test cases were already designed in the previous theory lesson. We monitored the students during this supervised practical and found that the actual testing based on the test-design was executed with ease.

The next 2 assignments had to be done from reading the specifications, design of the tests until executing the tests. For each assignment a compiled unit of software, with intentional bugs, was delivered. According to the theory of [van Merriënboer and Kirschner \[2007\]](#), we give less hints with every assignment, so that the difficulty level increases in a natural way. We did this by providing incomplete test-design templates where some properties are already in place.

Only a few questions were asked during the practical. We saw that the tests were neatly designed on paper and that the procedure was followed correctly. This gave us confidence that the lesson had arrived well. We asked a few students what they thought of the lecture and they said it was definitely an eye opener.

A week later we continued with lesson two. The main topic for this lesson is automated testing using JUnit. The lesson begins with the purpose of automated testing and the difference between black and white box testing. JUnit is briefly discussed, but we immediately move on to a practical situation. We introduce the procedural guide, as shown in Appendix A, and use it to code a test suite with JUnit for the specified method in Listing 4.3.

```

/**
 * @desc returns a dutch word-rated grade based on a numeric grade.
 *
 * @subcontract out of range grade {
 * @requires grade < 1 || grade > 10;
 * @signals (IllegalArgumentException) grade < 1 || grade > 10; }
 *
 * @subcontract inadequate {
 * @requires 1 <= grade < 5,5;
 * @ensures \result = "onvoldoende"; }
 *
 * @subcontract adequate {
 * @requires 5,5 <= grade < 7;
 * @ensures \result = "voldoende"; }
 *
 * @subcontract ample {
 * @requires 7 <= grade < 8;
 * @ensures \result = "ruim voldoende"; }
 *
 * @subcontract good {

```

```
* @requires 8 <= grade < 9;
* @ensures \result = "goed"; }
*
* @subcontract excellent {
* @requires 9 <= grade <= 10;
* @ensures \result = "uitmuntend"; }
*/
public static String getWordRating(double grade);
```

Listing 4.3: JML-specifications for *getWordRating()*

Step by step, we create a complete test suite based on this procedure. We constantly ask for input from the students and encourage discussions about how we are going to build the test suite. A striking detail was that in the boundary tests students want to take values that fall outside the input-domain of the subcontract. This has come up several times in the lesson and it also appears later in the elaborations and the 'thinking aloud' assignment of Milestone 6 that students do not realize that input values outside the pre-condition of the subcontract actually belong in another partition.

Another striking detail is that students are not yet fully aware of Exceptions. Some think you should throw an exception by returning it as returnvalue. Some are convinced that you really should throw an exception with catch. This lack of knowledge is probably due to the fact that this topic was only discussed in class two weeks ago and that a large part of the students is behind with the practice assignments. We also see this lack of knowledge in the elaborations and 'thinking aloud' sessions of Milestone 6, which were handed in 3 weeks later.

After designing and coding this test suite together with the students, following exactly the procedural guide, we explained the 'Milestone 6' assignment and encouraged students to start this assignment during the supervised practicum after class. Unfortunately, the class was scheduled on the Friday before the Christmas holidays, which meant that more than 50% of the students were absent during the practical. We have seen from the students who were present that the teaching material has arrived well. One problem students find difficult to solve is including the JUnit libraries in their project. Even though they learned how to include JavaFX two weeks earlier, it turns out that applying this knowledge to these other libraries is difficult.

In the last interviews and questionnaire, we asked students what they thought about the lessons and whether they experienced the lessons as difficult. For difficulty students could choose a value from one to five, where one stands for 'too easy' and a five stands for 'too difficult'. Based on 20 results, the mean and median were three with a standard deviation of 0.73. Based on this, we conclude that the difficulty level suits the intended target group.

With regard to the qualitative question of whether students have become more aware of code quality after the training, the majority of students only think about internal code quality and maintainability of code. We get a lot of references to the Software-design and Architecture lecture where we talked about maintainability of code. Concepts such as maintainability, code duplication, structure, single responsibility are fully discussed. Not a single student comes up with a concrete reference to the correctness of the written implementation code. A single student has remembered above all not to make assumptions. Another student said that through the testing lessons he became aware that writing correct code also means that you should cover invalid inputs and define exceptions for such cases and

also test those exceptions.

We think that the topic of code quality has been less lingering in the testing lesson because it has been overshadowed by the different concepts of code quality in the lecture Software design and architecture. It might be a good idea to revisit the concept of correctness in this software design and architecture lessons. It might also have been better to explicitly use the term 'correctness' in the phrasing of the questionnaire instead of code quality.

In the questionnaire we asked students for tips to improve the lessons about code-quality and testing. Overall students found the lessons useful, a single student mentioned that the large amount of specification-comments could scare some of the students. Students say the testing-subject is seen as boring, but because of the many examples and exercises in class it was 'flavored' well. The bonus-point in grading the group assignment was a good motivation to work on the 'Milestone 6' assignment. Testing feels too optional in later courses where lack of testing does not lead to failing the course or project. Some students suggest therefore to grade testing in later courses, projects and assignments to force testing.

4.3. STUDENTS BEHAVIOR

RQ3: What difference do we observe in coding behavior between students who use a test-first approach based on external specifications together with the procedural guidance, and students who do not use testing but have access to the specifications.

Analyzing the Milestone 6 assignment, it was noted that five of the twenty-five student couples designed test suites for all four implementations while only two were required and the other optional. The video analysis of the 'thinking aloud' session of one of these student couples, who already routinely approached testing, also includes the following quote "*Shall we define black-box tests for this assignment? We have to test it anyway!*", which has made it the subtitle of this graduation research. This statement clearly indicates that if the skills are present to make test-suites, creating a test-suite takes no extra time beyond trying manually to see if the implementation works. But it does continue to work as a guarantee for the correctness of the implementation.

We see in all student groups that the automated conversion of specifications to test suites is going reasonably well. From the preconditions of the specifications, the test values are chosen and the oracle is derived from the post-condition. The procedural guide is broadly followed. We can draw this conclusion because some aspects such as the AAA pattern (Arrange-Act-Assert) come out particularly well. But there are no indications that these students actually have the guide with them all the time. In the interviews, students indicate that the procedural guide is useful when learning the steps, but after performing this a few times during exercises, it goes automatically because this are fairly easy steps to master.

One aspect is not going well. Students quickly fall into making one test case per subcontract and forget to test the boundaries while they do talk about boundaries in the 'thinking-aloud' session. With other students we see that boundaries are taken outside the scope of the subcontract while the oracle is taken from within the scope of the subcontract. After making the implementation, these students find out that this test does not pass. After students reason together what is going wrong in the implementation, they find out after a while, without clear motivation, that they should just change the `assureTrue()` to `assure-`

False() in the test. Students now see that the test passes and they quickly move on to the next assignment.

With a single student couple we saw that the procedural guide was quickly consulted because there was a lack of clarity in the naming of the test method. Unfortunately, a while later this student couple changed the method names that matched the guide to different names because they liked that better. With the other couples we found no indication that the procedural guide was followed explicitly step by step, although we could see that the steps taken largely comply with the guide. Interviews with students also show that after the exercises, a large proportion of the students no longer feel the need to have the guide constantly next to them.

When creating the implementations that do not need test-suites, parts of the specifications are still copied and used almost one-to-one as implementation code. Students find the JML-specifications very useful for this, even if they do not need tests. A single student is convinced that the implementation based on the JML-specifications does not really need testing, because it fully complies with the specifications due to copying the conditions from the JML-specifications.

It is striking to see that the fumbling in the 'thinking-aloud' sessions has nothing to do with testing, but with the general programming skills that have not yet been developed to the intended level.

4.4. EXTERNAL CODE QUALITY

RQ4: What improvement do we see in the code correctness when using a test-first approach together with explicit informal specifications and the procedural guide.

In addition to the behavior of students, it is of course important to know whether a test-first approach still contributes to code correctness. The articles about testing in an educational setting are not exclusively positive as already concluded by [Desai et al. \[2008b\]](#), [Edwards and Shams \[2014\]](#), [Pancur et al. \[2003\]](#) and many others. We expect that by supplying explicit semi-formal JML specifications, the test suites made will score well on the quality aspect of completeness. We expect that the test suites created will not be limited to happy-path testing only, because the combination of the procedural guide and the JML-specifications also guides students through the non-happy-paths. We hypothesized that if the specifications in this study are given so explicitly that the implementation can be immediately derived from them, testing will add little to the quality aspect of code correctness of the implementation. The results show us that we are wrong about this, as we discover later in 4.4.5 after we cover the analysis of the submitted assignments in the next sections.

For the four methods that had to be coded for the 'Milestone 6' assignment, we will analyze the results method-by-method.

4.4.1. *NumericRangeTools.isValidPercentage()*

This method excels in its simplicity. The specifications of this method can be found in Listing 4.4. This assignment was created with the focus on being able to easily generate a complete test set. Due to the simplicity, it is also obvious that the implementation can be easily implemented by a one-liner. We therefore expect high-quality submissions from the students.

```
public class NumericRangeTools {
```

```

/**
 * @desc Validates if the input is within range of 0-100%
 *
 * @subcontract value within valid range {
 *   @requires 0 <= percentage <= 100;
 *   @ensures \result = true;
 * }
 *
 * @subcontract value out of range low {
 *   @requires percentage < 0;
 *   @ensures \result = false;
 * }
 *
 * @subcontract value out of range high {
 *   @requires percentage > 100;
 *   @ensures \result = false;
 * }
 *
 */
public static boolean isValidPercentage(int percentage);
}

```

Listing 4.4: JML-specifications for *isValidPercentage()*

For this assignment, there are fourteen student couples (56%) who submitted the implementation with a test suite. The remaining eleven student couples (44%) only made the implementation of this method.

To assess correctness on the implementation, we tested the submitted implementation against a validated teacher test set. Of the fourteen student couples who also made a test set, twelve student couples turned out to have made a completely correct implementation. This is 85.7% of the group who wrote a test set for this method. Of the eleven student couples that chose not to make a test set, eight student couples turned out to have made a completely correct implementation. This is 72.2% of the non-testing group. In order not to let students with really bad programming skills obscure the results, we also take the median of the correctness. Both groups score here the expected 100% correctness for the implementation.

After the implementations, we also reviewed the test sets. We assess the test sets on two quality aspects, namely correctness and completeness.

To determine the correctness of the test sets, we test against a validated teacher implementation, we also test against an inverted teacher implementation to detect tests that always pass. Of the fourteen test sets made, ten (71.4%) sets test correctly. Surprisingly, there are two student couples with good implementation but failing tests. When we look at the recorded 'Thinking aloud' session, we see that these groups do pre-create test sets, they do make the implementation, but forget to test the implementation against their test set. We conclude from this behavior that testing with these students is not seen as a tool to ensure the correctness of the implementation, but merely as a task from school to be completed.

To assess the completeness of the test sets, we use mutation testing on the teacher implementation. Any mutation made to the teacher implementation must fail at least one test from the submitted test set. We see that eight of the fourteen test sets (57.1%) are complete. On average we see that 81.4% of the mutations are detected, the median on mutation detection for this method is 100%.

4.4.2. *DateTools.validateDate()*

To increase the difficulty, we have put together a method that should be able to detect a valid date from the combination of day, month and year. The specifications for this method are in Listing 4.5. What makes this assignment more difficult is the amount of edge cases that can be found in the different subcontracts. We expect that there are only a few students who make so many tests that the edge cases are covered. We are more optimistic about implementation, the JML specifications are so explicit that the implementation can be derived from these specifications one-to-one.

```
public class DateTools {
    /**
     * @desc Validates a given date in the form of day, month and year
     * is valid.
     *
     * @subcontract 31 days in month {
     *   @requires (month == 1 || month == 3 || month == 5 || month == 7 ||
     *             month == 8 || month == 10 || month == 12) && 1 <= day <= 31;
     *   @ensures \result = true;
     * }
     *
     * @subcontract 30 days in month {
     *   @requires (month == 4 || month == 6 || month == 9 || month == 11) &&
     *             1 <= day <= 30;
     *   @ensures \result = true;
     * }
     *
     * @subcontract 29 days in month {
     *   @requires month == 2 && 1 <= day <= 29 &&
     *             (year % 4 == 0 && (year % 100 != 0 || year % 400 == 0));
     *   @ensures \result = true;
     * }
     *
     * @subcontract 28 days in month {
     *   @requires month == 2 && 1 <= day <= 28 &&
     *             (year % 4 != 0 || (year % 100 == 0 && year % 400 != 0));
     *   @ensures \result = true;
     * }
     *
     * @subcontract all other cases {
     *   @requires no other accepting precondition;
     *   @ensures \result = false;
     * }
     *

```

```
*/  
public static boolean validateDate(int day, int month, int year);  
}
```

Listing 4.5: JML-specifications for *validateDate()*

For this assignment, there are sixteen student couples (64%) who submitted the implementation with a test suite. The remaining nine student couples (36%) only made the implementation of this method.

To assess correctness on the implementation, we again tested the submitted implementation against a validated teacher test set. Of the sixteen student couples who also made a test set, fourteen student couples turned out to have made a completely correct implementation. This is 56.3% of the group who wrote a test set for this method. Of the nine student couples that chose not to make a test set, five student couples turned out to have made a completely correct implementation. This is 55.6% of the non-testing group.

With the teacher test set we detected 8.2% failing test cases on the implementations that were supplied with the test set. On the implementations that were submitted without a test set, we detect 13.6% failing tests. So here we see a difference in the correctness of the implementations made. Now it is true that the implementation using the JML-specifications was not that difficult. The majority of the students provided a completely correct implementation. The determined median on failing test cases across all elaborations is 0%.

After the implementations, we again reviewed the test sets. We assess the test sets on two quality aspects, namely correctness and completeness again.

To determine the correctness of the test sets, we test against a validated teacher implementation, we also test against an inverted teacher implementation to detect tests that always pass. Of the sixteen test sets made, thirteen (81.3%) sets test correctly. Again there is a student couple with good implementation but failing tests on their own implementation. When we look at the recorded 'Thinking aloud' session, we see that this couple thinks that they should also write tests that intentionally fail. This could not happen if the procedural guide had been followed step-by-step. We can certainly conclude for this student couple that they certainly do not use the step-by-step plan for this assignment.

To assess the completeness of the test sets, we use mutation testing on the teacher implementation again. This method has a lot of mutations therefore we expect a low percentage on completeness of the test set. After analysis of the mutation test, we can conclude that no student couple has supplied a complete test set in which all mutations are detected. The average test completeness over all submitted tests is 72.1%, the median is 76%. We think this is not a bad score after all.

4.4.3. *MailTools.validateMailAddress()*

The next assignment is a validation assignment in which a string must be validated on a correct format. Students have not yet been introduced to regular expressions, so the JML specifications are based on the '*String.split()*' method. The specifications can be found in Listing 4.6. Email addresses are only valid if they consist of one single domain name with a top-level domain. Email addresses such as 'mailbox@subdomain.domain.tld' are invalid according to these specifications. We are curious if students make assumptions and still accept additional subdomains in the implementation of this validation method.

```

public class MailTools {
    /**
     * @desc Validates if mailAddress is valid. It should be in the form of:
     *     <at least 1 character>@<at least 1 character>.<at least
     *     1 character>
     *
     * @subcontract no mailbox part {
     *     @requires !mailAddress.contains("@") ||
     *         mailAddress.split("@")[0].length < 1;
     *     @ensures \result = false;
     * }
     *
     * @subcontract subdomain-tld delimiter {
     *     @requires !mailAddress.contains("@") ||
     *         mailAddress.split("@")[1].split(".").length > 2;
     *     @ensures \result = false;
     * }
     *
     * @subcontract no subdomain part {
     *     @requires !mailAddress.contains("@") ||
     *         mailAddress.split("@")[1].split(".")[0].length < 1;
     *     @ensures \result = false;
     * }
     *
     * @subcontract no tld part {
     *     @requires !mailAddress.contains("@") ||
     *         mailAddress.split("@")[1].split(".")[1].length < 1;
     *     @ensures \result = false;
     * }
     *
     * @subcontract valid email {
     *     @requires no other precondition
     *     @ensures \result = true;
     * }
     */
    public static boolean validateMailAddress(String mailAddress);
}

```

Listing 4.6: JML-specifications for *validateMailAddress()*

For this assignment, there are sixteen student couples (64%) who submitted the implementation with a test suite. The remaining nine student couples (36%) only made the implementation of this method.

To assess correctness on the implementation, we again tested the submitted implementation against a validated teacher test set. Striking fact is that only one student couple managed to deliver a correct implementation. This couple also made a test set for this method. Almost all students go wrong with the fact that in this method only a single domain name with tld is valid. Even the tip in the description of the method '*<at least 1 character>@<at*

least 1 character>.<at least 1 character>' didn't help them.

Upon closer inspection of the elaborations, it appears that many students have copied '*split(".")*' from the specifications into the implementations. However, the '.' in this split method is not a literal string but a regular expression, meaning any character. If students really want to split on the character '.' then they will have to escape this character with a backslash like this: '\\.'

The fact that '*split()*' works with regular expressions in Java was unforeseen when drafting the assignment. The meaning in the specifications is split on the character '.'. This is not a bad thing in itself, because we can now analyze what students do if the specification contains something that cannot be transferred one-to-one in Java.

In the 'Thinking aloud' sessions we see that many students do not see this unintended effect, because they do not test correctly. The students who do notice the unintended effect of the '*split()*' method struggle a lot before they find the solution on the internet.

If we look at the percentage of teacher tests that fail on implementations, we see that in the group of students who also made test sets, 28% of the tests fail, with a median of 21.1%. Of the group of students who did not write a test set for this method, on average 56% of the teacher tests fail with a median of 68.4%.

Although the implementation caused problems for most students due to the unforeseen regular expression in the '*split()*' method, students could of course have made a good test set. After an analysis of the test sets with the teacher implementation and a negation of this implementation, we see that nine of the sixteen student couples submitted a correct test set. A total of 95 tests were written by these students. Of these tests, six (6.3%) fail on the validated teacher implementation.

4.4.4. *PostalCode.formatPostalCode()*

The '*formatPostalCode()*' method adds a different kind of complexity. Not only does this method perform validation, but it must also properly format and return a given string. Due to the additional complexity, we expect that students may write more test cases to ensure the correctness of their implementation. The specification of this method can be found in Listing 4.7

```
public class PostalCode {

    /**
     * @desc Formats the input postal code to a uniform output in the form
     * nnnn<space>MM, where nnnn is numeric and > 999 and MM are 2
     * capital letters.
     * Spaces before and after the input string are trimmed.
     *
     * @subcontract null postalCode {
     *   @requires postalCode == null;
     *   @signals (NullPointerException) postalCode == null;
     * }
     *
     * @subcontract valid postalCode {
     *   @requires Integer.valueOf(postalCode.trim().substring(0, 4)) > 999
     *   && Integer.valueOf(postalCode.trim().substring(0, 4)) <= 9999
```

```

*      && postalCode.trim().substring(4).trim().length == 2
*      && 'A' <=
*          postalCode.trim().substring(4).trim().toUpperCase().charAt(0)
*          <= 'Z'
*      && 'A' <=
*          postalCode.trim().substring(4).trim().toUpperCase().charAt(0)
*          <= 'Z';
*      @ensures \result = postalCode.trim().substring(0, 4) + " " +
*                  postalCode.trim().substring(4).trim().toUpperCase()
*  }
*
*  @subcontract invalid postalCode {
*      @requires no other valid precondition;
*      @signals (IllegalArgumentException);
*  }
*
*/
public static String formatPostalCode(String postalCode);
}

```

Listing 4.7: JML-specifications for *formatPostalCode()*

For this assignment, there are fifteen student couples (60%) who submitted the implementation with a test suite. The remaining ten student couples (40%) only made the implementation of this method.

No single student couple has managed to implement this method correctly. Striking about the elaboration of this method is the fact that lots of students have not yet mastered the skills of throwing exceptions and testing these exceptions, even though this was already discussed in class weeks ago. In the 'Thinking Aloud' session, we see them struggle to get the exceptions working.

We do see major differences between the elaborations of students who have written tests in advance and students who have only made the implementation. Of the teacher test set, which consists of 20 test cases, an average of 28% (median 10%) of the tests fail in students who have written tests in advance, compared to 37% (median 25%) of the test cases in student couples who have not written any tests for their implementation.

4.4.5. OVERALL RESULTS

If we look at the general picture of code correctness of the implementation, summarized in Table 4.1, we can conclude that a test-first approach certainly contributes to improving code correctness even when students without tests have the same JML-specifications at hand.

If we combine data on all 4 assignments together for the 25 student couples, we see at $n=100$ (25 couples \times 4 assignments) an average of 17% failing teacher tests with students who use test-first together with the procedural guide and with explicit JML-specifications. Students who have the same JML-specifications but choose not to write tests score an average of 31% failing teacher tests.

If we look at the achieved completeness of the students' test sets, we see that for the assignments *validateDate()* and *isValidPercentage()* on average 73% of the bugs are found

method name	students using test-first		students without testsets	
	average	median	average	median
<i>isValidPercentage()</i>	4%	0%	17%	0%
<i>validateDate()</i>	8%	0%	14%	0%
<i>validateMailAddress()</i>	28%	21%	56%	68%
<i>formatPostalCode()</i>	28%	10%	37%	25%

Table 4.1: Tests failing percentage on implementations using a reference test-set

(42 mutations of the validated teacher implementations). [Edwards and Shams \[2014\]](#) found in their research that students are on average only able to find 13.6% of the faults.

This remarkable difference can be found in the added combination of procedural guide and the explicit JML-specifications. This combination ensures that students do not stray into happy-path testing but procedurally leads them to test all input partitions.

4.5. STUDENTS' (UN)WILLINGNESS TO INVEST TIME FOR QUALITY

In the interview and the questionnaire we asked whether and where students used the procedural guide. Students could choose between the following answers:

- a. Yes, during the lesson assignments and CodeCademy "Milestone 6" assignment.
- b. Yes, only during the assignments in the lesson.
- c. Yes, only during CodeCademy "Milestone 6" assignment.
- d. Yes, a few times, after that the steps were completely clear and I didn't need it anymore.
- e. No, i did not use the guide.
- f. Procedural guide for testing? I have no idea what it is about.

The results are shown in [Figure 4.2](#):

An interesting fact is that during the analysis of the 'thinking aloud' recordings there are no indications that the guide is physically present, but for the most part the steps are followed correctly.

One student couple even designed the test cases in the provided template ([Figure 4.1](#)) for coding the tests. Which was only used during the first practical, not to be hindered by the JUnit framework complexity, to think about suitable test values.

The interviews also show that students find the process fairly easy and logical. After practicing with the guide, there is no longer any need to keep the guide with you. One student says they have set the guide aside as a cheat sheet in case he doesn't remember. The two students who no longer recognize the procedural guide also indicate that the difficulty of the test lessons is 'high' in a previous question. Maybe because of the difficulty or lack of motivation they dropped out to take the lessons seriously.

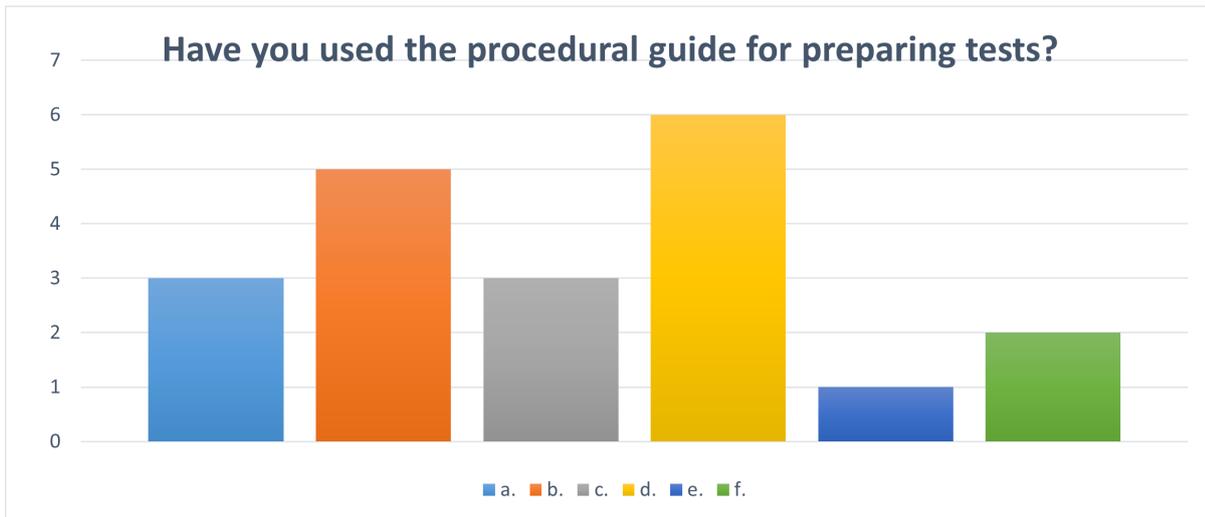


Figure 4.2: Usage of the procedural guide

When asked whether the procedural guide was helpful or not, 16 of the 20 students indicated that they found the guide useful, two students were neutral, and two students indicated that they did not see any added value in the guide. Overall, students like the procedural guide because it provides an overview, it helps to better understand the transformation from specification to code, it makes clear in which order you should approach the conversion, it provides systematic support and so you do not forget anything. Students who do not find the guide helpful say the process is simple enough and if they do not understand it, they will look for an instructional video on the internet to get a demonstration. This creates a fear among us as teachers. Who judges the quality of these youtube knowledge clips? We will take this comment into account, perhaps that we can record a knowledge clip ourselves and put it in an accessible place for students.

The interviews already showed that students do not recognize the term specifications, whether or not supplemented with the term JML. They are still familiar with the term sub-contracts and in an interview they immediately relate it to testing boundaries. From this we conclude that knowledge about partitioning and boundary values is present, but that the terms JML and specifications have been forgotten. For the questionnaire we included therefore an example of a JML-specification.

In the questionnaire we asked students whether the JML-specifications helped with making the assignment. They could choose from the following answers:

- a. Yes, for both the coding of the tests and the implementations.
- b. Yes, for coding the tests.
- c. Yes, for coding the implementations.
- d. No, it didn't help me.
- e. JML or (sub)contracts? I have no idea what it's about, I must have missed that lesson.

The results are shown in Figure 4.3:

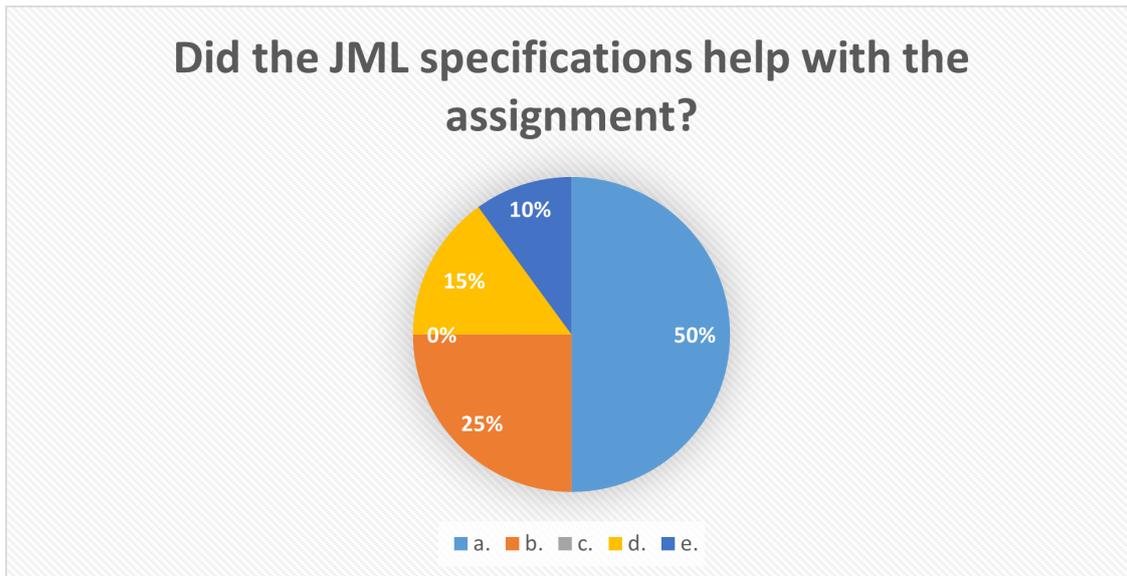


Figure 4.3: Helpfulness of the JML specifications

It was no surprise to see that the majority of students choose helpful in writing the tests and the implementation. After all, we already saw in the analysis of the 'Thinking-aloud' assignment that without exception the pre- and post-conditions from the JML were copied and pasted into the tests and implementations.

As for the qualitative elucidation of this question, students answer that JML-specifications help to provide clarity about how the implementation should work, it provides guidance in determining the boundaries to be tested, it was even so clear that the specifications could be mapped one-on-one in the tests and implementation. The students who did not find JML-specifications helpful indicate that they find it a strange concept to 'code' the JML-specifications and then code exactly the same in the tests and after one time again in the implementation. They indicate that they do not see the added value above an informal comment in descriptive text prior to the method. Of course we could do something with this, such as coming up with an assignment where the specifications cannot be copied one-to-one to the tests or to the implementation. On the other hand, it is precisely our intention to make the conversion of specifications into tests and implementations as accessible as possible for these novice programmers.

Furthermore, the questionnaire asked whether students would be willing to compose JML-specifications themselves from a problem statement in a natural language. The students could tick multiple statements and the statements were as follows:

- a. Yes of course, it definitely improves the code quality because it makes me think more about the problem.
- b. Yes, I would like to do that if there is also a handy step-by-step plan for this.
- c. Yes, but only if it is part of a graded review.
- d. Yes, but not in such a strict JML way.
- e. No, writing tests already gives enough specifications to code the method.

- f. No, with the same effort I immediately code such a method.
- g. No, it takes me too much time.

The results are getting a bit more surprising. We hypothesized that students who see the benefit of explicitly composed JML specifications themselves will be willing to voluntarily compose them. The answer to this question shows that less than half, with or without the help of a procedural guide, would be willing to do this. Six students answered that they only do this if it is actually graded, eight students indicate that they would specify it, only in a less formal way, eight answers indicate that they do not want to do it because it takes too much time. Composing semi-formal specifications is therefore considered as cumbersome and time-consuming. The results are shown in Figure 4.4:

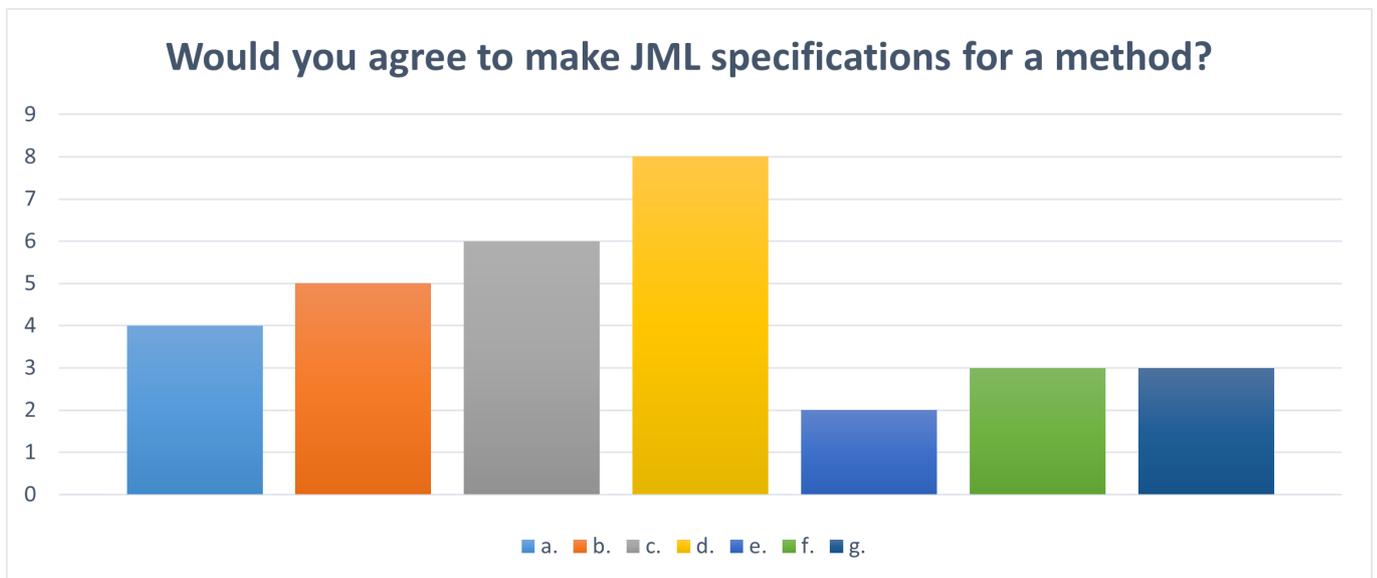


Figure 4.4: Willingness to make JML specifications

The follow-up question, "I am willing to write automated tests in the future because...", where students can again tick several statements, shows once again that the willingness to make tests is only present in a large part of the students if there is a graded assessment against it, or that writing automated tests is explicitly requested by the client. The following statements could be chosen:

- a. It increases the quality of my code.
- b. Only if it is part of a review.
- c. Only if the client asks for it.
- d. No, I do not because manual testing or trying it out is just as effective.
- e. No, I do not because it takes too much time.
- f. No, I do not because it doesn't add anything to the code quality

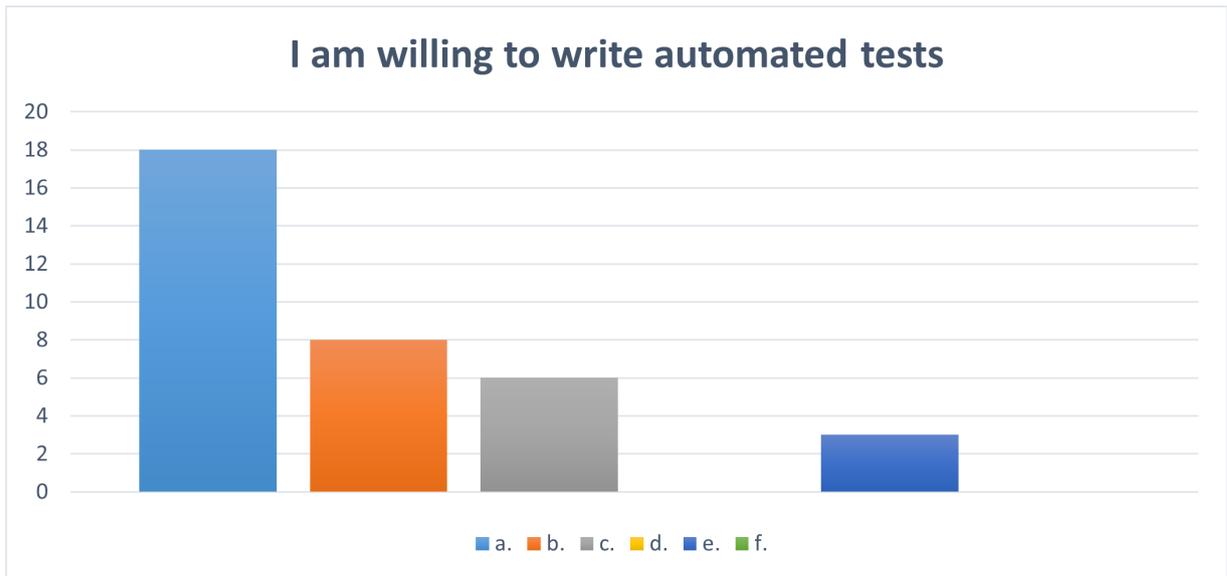


Figure 4.5: Willingness to make automated tests

Answers in the questionnaire are shown in Figure 4.5:

In these results we notice a lack of intrinsic willingness to write tests. Scatalon et al. [2017a] wrote exactly our feeling at this stage: *"There is a widespread agreement that novice programmers are reluctant to conduct software testing [refs]. When it is not made compulsory by the instructor, students tend not to do it [ref]. Usually, to force students to conduct testing, part of their grade depends on it. However, this imposition can be increasing even more students' resistance, because while they do not really appreciate the value of testing, they can continue see it only as an unnecessary burden [ref]."*

We assume that the intention to write tests is often there, but that this is often postponed to a test-last approach, after which it never is realised because there was no time left. Of course this is a fairly natural phenomenon, after all, coding software is never finished, as a software engineer you abandon the software at some point because the budget is exhausted and it meets the customers requirements. There is always an opportunity to make the software even better, add even more functionality for the customer and thus further postpone the creation of tests. This happens exactly with our students.

At the time the interviews and the questionnaires took place, the students are busy creating applications for external clients. We asked them how they handle testing and asked them to tick one or more of the following statements:

- a. We as a team have taken the initiative to write tests in advance (before implementation).
- b. We as a team have taken the initiative to write tests afterwards (after the implementation).
- c. As a team, we have been instructed to write tests in advance (before implementation).
- d. As a team, we were commissioned to write tests afterwards (after the implementation).

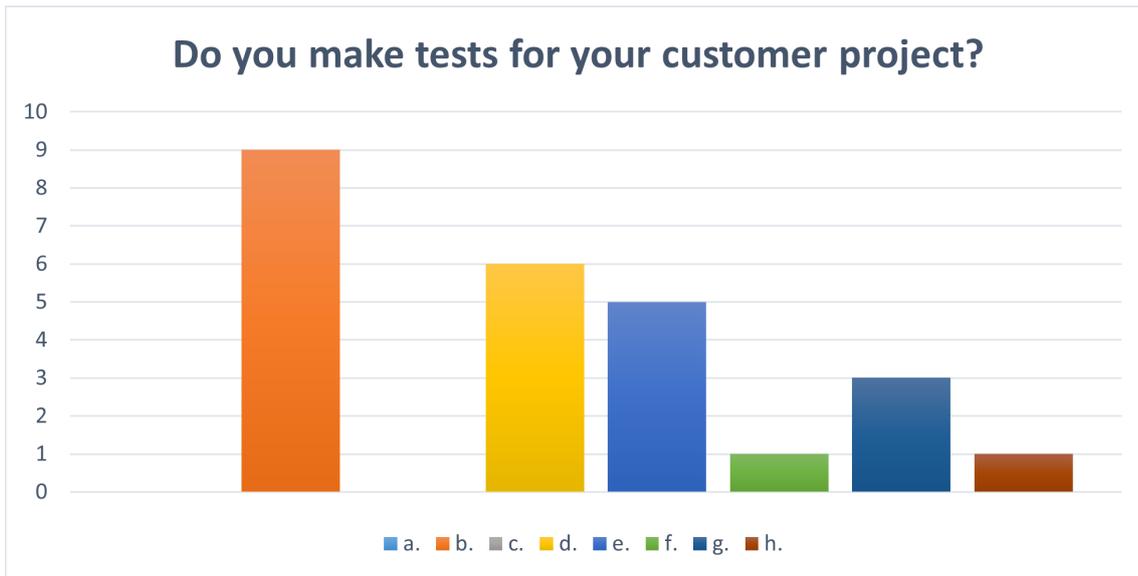


Figure 4.6: Testing for an external customer

- e. We do not write automated tests because we don't have enough time.
- f. We do not write automated tests because we are not graded on them.
- g. We do not write automated tests because we don't think it is necessary.
- h. We do not write automated tests because the client does not ask for it.

The reactions all seem to result in postponing the making of tests. This procrastination does not only come from students themselves, but also from customers who prefer quantity (functionality) over quality. The results are shown in Figure 4.6:

When the projects were handed over to the companies, a single group appeared to have made some tests. The majority have not gotten around to testing despite their intention to do so.

4.6. OTHER OBSERVATIONS

During the analysis of the 'thinking aloud' assignment, we frequently saw students struggling with the same situations.

- Getting the JUnit test framework working.
- Working with Exceptions. Concepts such as throwing and catching exceptions are not clear to students.
- Debugging and breakpoints. Even though we've insisted on using the debug tooling in the IDE and while we demonstrate algorithms in debug mode step by step in every lesson, not a single student uses debugging as an analytic tool if the output unexpectedly fails. Instead, students keep trying different alternative implementations.

The lack of skills could be due to the current Covid-19 situation. Students are not physically present at school for lessons, have little or no bond with fellow students and teachers. From conversations that lecturers have with students in their position as study career counselor, we hear that many students have difficulty developing a regular study rhythm and motivating themselves to complete assignments. As teachers in the 'Programming 2' course, we also saw that students were lagging considerably behind with the practice assignments.

During the exam for the course 'Programming 2', we as teachers saw that students frequently use StackOverflow for their programming assignment, while the knowledge from the course 'Programming 2' is actually assumed to be basic knowledge. Students seem to prefer copying and pasting the code and get the method working through experimentation rather than understanding how the concept works and writing code themselves. Students do not realize that they ultimately fail the exam due to lack of skills and therefore lack of time.

It is also remarkable that students do not recognize a boolean expression as such. As a result, we often come across code constructs where students apply a boolean expression as condition within an if-statement, and then set another boolean variable within the execution paths for if and else. An example is shown in Listing 4.8.

```
public boolean someFunction() {
    boolean returnValue = false;
    if(value > 5) {
        returnValue = true;
    } else {
        returnValue = false;
    }
    return returnValue;
}
```

Listing 4.8: Code construct often created by students

When we show students the easy single liner as in shown in Listing 4.9 students tend to look surprised that this is even possible.

```
public boolean someFunction() {
    return value > 5;
}
```

Listing 4.9: Code construct what teachers would like to see

5

CONCLUSION

How do explicit specifications, a test-first approach, and procedural guidance to convert specifications to tests, help to improve code correctness among students?

If we fall back on this research question of this study, we can answer this question in three parts, namely:

- **How do explicit specifications help to improve code correctness among students?**

We can conclude that students find explicit JML-specifications useful, at least as long as they are made available for the exercises, and the specifications contribute to the code correctness.

The interviews and questionnaires show that students no longer recognize the terms JML or specifications after five months, despite that, they are still aware of the various contracts that reveal the inner working of the software-unit and that they have to test the boundaries of these contracts well.

Students also remember not to make assumptions about how the software should work. Explicit specifications always seem to be complete specifications for students, while this does not necessarily have to be the case of course. This is still overlooked by students, but as teachers we accept this because students are still novice programmers.

The JML specifications give students a guideline in designing the tests, it gives them an indication of which tests are needed and also helps students to code the implementation. This convenience is recognized by many students who participated in the interviews and questionnaire. We also saw in the 'thinking-aloud' sessions that students really use the JML-specifications as a guideline for designing the tests and also for coding the implementation.

It is a pity that, despite the benefit students get from the specifications, students themselves indicate that they do not want to invest the time in drawing up the specifications so explicitly. Some students indicate that they may want to do this in an informal commentary style, but not in such a formal JML way as we have provided the specifications. Even if we indicate that we will provide a procedural guide for writing such specifications, the willingness still seems limited. This could be due to the fact that students think they are only drawing up the specifications for themselves.

Perhaps we, as teachers, should put the drafting of explicit JML-specifications into a practical situation where one student draws up explicit specifications from the requirements and shares them as a way of communication with another student who codes the tests or implementation. In this way, explicit specification is experienced as more useful and we get more willingness among the students

- **How does a test-first approach help to improve code correctness among students?**

The test-first approach certainly helps in increasing the correctness of the code. We previously hypothesized that with such fully explicit specifications we expected little room for a quality-enhancing effect on code correctness. The technical analysis in section 4.4.5 of the 'Milestone 6' assignments proves that the correctness of the implementations for which test suites have been written score significantly higher than implementations for which no tests have been written. From the 'thinking- aloud' sessions, we see that students also make extensive use of the JML-specifications for implementations, even though they do not create test suites for this.

- **How does a procedural guide help to improve code correctness among students?**

The procedural guide has an indirect positive effect on the code correctness of the implementations. We have rarely seen students consult the guide physically in the 'thinking-aloud' sessions, but it appears from these recordings and the submitted elaborations of the assignments that the steps have been completed successfully for the most part.

The interviews and questionnaire show that students certainly found the procedural guide helpful during the learning process. However, they point out that the process of designing the tests and writing the JUnit tests is so simple that the guide is only needed to practice it a few times. What is unfortunate, and is apparent from the 'thinking-aloud' recordings, is that students who are stuck for a while do not return to the procedural guide.

5.1. DISCUSSION

In general, we can say that these three components together certainly increase the quality aspect of code correctness. However, what also emerges from the research is that, even though students know that the quality of their work is improving, they seem not to be willing to invest the time and effort in testing to produce better code. The willingness to apply testing increases if we make testing mandatory for an assignment, but if testing is only a small part of the grade of an assessment, testing is often omitted at the cost of a small amount of points.

We try to find a reason for this evasive test behavior. When looking at the coding behavior of students, we can endorse the research by Edwards [2004b] in which he states that students get stuck in trial-and-error problem solving. We see exactly the same behavior in our students when working on projects and assignments. We see that students often start from an existing code snippet, found on 'stack overflow' or from another source. Students do not tend to use debugging functionalities but randomly change a few lines of code and see if the output of the program is acceptable afterwards. This could be due to the fact that programming skills are still so underdeveloped that students already think it is good if software already does something that is in the right direction as already concluded by Kolikant

[2005]. Based on this, we hypothesize that as students become more proficient in programming, they also become more aware of code quality and specifically its correctness.

Providing explicit JML-specifications has suppressed 'stack overflow' copy-paste behavior. This can certainly be seen as something positive. It seems that the specifications give students a good starting point for coding, where they otherwise look for a code snippet that does about what is expected.

For students, software seems acceptable when the program does roughly what it is supposed to do. This behaviour confirms to the hypothesis of [Kolikant \[2005\]](#) where she states that students are easily satisfied with incorrect software if some output seems reasonable. According to [Kolikant \[2005\]](#) this is because students' understanding of correctness is different from that of professionals. Also the testing methods students apply are found inadequate by professionals.

Despite the fact that we anticipated this behavior in advance by using the concept of code correctness as a common thread through the lessons, we have not been able to get students to take testing so seriously that they independently see the importance of correct specifications and associated tests to ensure correctness as far as possible.

Fortunately, a bonus point for the exam convinced half the student population to participate in the 'Milestone 6' assignment. It might obscure the results a bit because it is mainly the better students who participated in this assignment. However, this does not affect the validity of the study in our opinion. Normally, a substantial part of the student population will leave before february 1st. because they cannot keep up with the programming skills. The number of students who drop out in the first year is approximately 50%, which roughly corresponds to the part who did not participate in the 'Milestone 6' assignment.

Nevertheless, we do not consider our research and adapted educational program any less valuable. It appears from the interviews and questionnaire that six months later certain knowledge about testing and code quality, such as contracts and boundary testing, has remained. What has failed is that students start testing on their own, driven by the fact that they have to deliver quality code.

The coding skills of the students seem somewhat lower than in previous years, but that can be explained by the fact that there were only a few physical classes due to COVID-19. Many students struggled to find motivation to study during the lockdown. Of course, this applies to some students to a greater or lesser extent than to others. Since we have seen that many students fell short with their skills during the 'Milestone 6' assignment, we expect that the results will be more positive during a non-lockdown scenario with physical lessons.

For the behavioral analysis of our students, our planning was to largely rely on semi-structured interviews together with the 'Thinking aloud' assignment. A considerable number of students were initially willing to participate in the student interviews. However, when we wanted to make the appointments concrete, many students dropped out because they were too busy with other school assignments. With the information that we were able to retrieve from the six student interviews we conducted, we drafted a questionnaire in which we could present the most striking matters from the interviews to a larger group of students and ask for a response. To increase the willingness to participate in this questionnaire, we raffled a gift voucher among the participants. In the end, twenty students completed the questionnaire. With which we have been able to substantiate certain points from the in-

interview more strongly. We do not expect to receive unreliable answers because of the gift voucher raffle because the open questions of all participants were filled in with great detail.

Finally, we would like to add to the fact that during student project work in which students really make software for companies, the quantity of requirements, the making of functionality, always takes precedence over quality and therefore testing. It is not the teachers who direct this behavior, but especially the professionals from the business community who make testing subordinate. This could be because companies often have students create greenfield prototypes for which some bugs are acceptable and do not require testing. We as educators need to become more controlling in this and ensure that automated testing becomes part of the coding of functionality, even if students make prototypes for companies.

5.2. FUTURE WORK

As teachers, our ultimate goal is of course to properly train future professionals. The teaching of code quality and the adequate testing of developed software is indispensable in this regard. Although we are convinced that conducting the lessons on code quality, specifications and testing in this way positively contributes to the development of the students, the hurdle of intrinsic testing behavior has not yet been overcome.

We think that [Kolikant \[2005\]](#) has already defined a good starting point to analyze where the test-avoiding behavior comes from. It seems obvious that when teaching a good definition of correctness, we should encourage students in intrinsic testing behaviour. On the other hand, with the new set-up of the teaching material, we have already made correctness very explicit and frequent, which still does not motivate students to start testing themselves.

Our hypothesis that when students become more proficient in programming, they also become more aware of code quality and specifically its correctness is worth investigation. If our hypothesis is correct we should accept that beginning students do not yet have the ability to code very quality-consciously. We should investigate whether there is a difference in quality awareness or willingness to test in later years after coding skills are better developed.

Also it is worth investigating at what stage of the learning progress providing knowledge about quality and testing is most efficient in education. Now we apply this knowledge at the same time basic programming skills are learned. We should investigate whether this is the best time, or that it is better to postpone this teaching material about quality and testing to a later moment when the basic programming skills are more developed.

BIBLIOGRAPHY

Kent Beck. Extreme Programming Explained: Embrace Change. Addison-Wesley Publishing Company, 1999. 4

Kent Beck. Test-driven development: by example. Addison-Wesley Professional, 2003. 4

Thirumalesh Bhat and Nachiappan Nagappan. Evaluating the efficacy of test-driven development: Industrial case studies. In Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering, ISESE '06, page 356–363, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595932186. doi: 10.1145/1159733.1159787. URL <https://doi.org/10.1145/1159733.1159787>. 5

Lex Bijlsma, Niels Doorn, Harrie Passier, Harold Pootjes, and Sylvia Stuurman. How do students test software units? In 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET), pages 189–198, 2021. doi: 10.1109/ICSE-SEET52601.2021.00029. 1

Wilson Bissi, Adolfo Gustavo Serra Seca Neto, and Maria Claudia Figueiredo Pereira Emer. The effects of test driven development on internal quality, external quality and productivity: A systematic review. Information and Software Technology, 74:45 – 54, 2016. ISSN 0950-5849. doi: <https://doi.org/10.1016/j.infsof.2016.02.004>. URL <http://www.sciencedirect.com/science/article/pii/S0950584916300222>. 5

Chetan Desai, David Janzen, and Kyle Savage. A survey of evidence for test-driven development in academia. ACM SIGCSE Bulletin, 40(2):97–101, 2008a. 1

Chetan Desai, David Janzen, and Kyle Savage. A survey of evidence for test-driven development in academia. ACM SIGCSE Bulletin, 40(2):97–101, June 2008b. ISSN 0097-8418. doi: 10.1145/1383602.1383644. URL <https://dl.acm.org/doi/10.1145/1383602.1383644>. 3, 27

Niels Doorn. How can more students become test infected, 07 2018. 3

Stephen H. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. SIGCSE Bull., 36(1):26–30, March 2004a. ISSN 0097-8418. doi: 10.1145/1028174.971312. URL <https://doi.org/10.1145/1028174.971312>. 1, 2

Stephen H. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. In Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education, SIGCSE '04, page 26–30, New York, NY, USA, 2004b. Association for Computing Machinery. ISBN 1581137982. doi: 10.1145/971300.971312. URL <https://doi.org/10.1145/971300.971312>. 42

- Stephen H. Edwards and Zalia Shams. Do student programmers all tend to write the same software tests? In Proceedings of the 2014 Conference on Innovation; Technology in Computer Science Education, ITiCSE '14, page 171–176, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328333. doi: 10.1145/2591708.2591757. URL <https://doi.org/10.1145/2591708.2591757>. 3, 27, 34
- Yifat Ben-David Kolikant. Students' alternative standards for correctness. In Proceedings of the First International Workshop on Computing Education Research, ICER '05, page 37–43, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930434. doi: 10.1145/1089786.1089790. URL <https://doi.org/10.1145/1089786.1089790>. 42, 43, 44
- Laura Marie Leventhal, Barbee Eve Teasley, and Diane Schertler Rohlman. Analyses of factors related to positive test bias in software testing. International Journal of Human-Computer Studies, 41(5):717 – 749, 1994. ISSN 1071-5819. doi: <https://doi.org/10.1006/ijhc.1994.1079>. URL <http://www.sciencedirect.com/science/article/pii/S1071581984710792>. 3
- Tilman Michaeli and Ralf Romeike. Addressing Teaching Practices Regarding Software Quality: Testing and Debugging in the Classroom. In Proceedings of the 12th Workshop on Primary and Secondary Computing Education, pages 105–106, Nijmegen Netherlands, November 2017. ACM. ISBN 978-1-4503-5428-8. doi: 10.1145/3137065.3137087. URL <https://dl.acm.org/doi/10.1145/3137065.3137087>. 1, 3
- Glenford J. Myers, Corey Sandler, and Tom Badgett. The art of software testing. John Wiley & Sons, Hoboken and N.J, 3rd ed edition, 2012. 2, 3
- M. Pancur, M. Ciglaric, M. Trampus, and T. Vidmar. Towards empirical evaluation of test-driven development in a university environment. In The IEEE Region 8 EUROCON 2003. Computer as a Tool., volume 2, pages 83–86 vol.2, 2003. 5, 27
- Nadia Polikarpova, Carlo A Furia, Yu Pei, Yi Wei, and Bertrand Meyer. What good are strong specifications? In 2013 35th International Conference on Software Engineering (ICSE), pages 262–271. IEEE, 2013. 6, 8
- Giuseppe Scanniello, Simone Romano, Davide Fucci, Burak Turhan, and Natalia Juristo. Students' and professionals' perceptions of test-driven development: A focus group study. In Proceedings of the 31st Annual ACM Symposium on Applied Computing, SAC '16, page 1422–1427, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450337397. doi: 10.1145/2851613.2851778. URL <https://doi.org/10.1145/2851613.2851778>. 5
- Lilian Passos Scatalon, Ellen Francine Barbosa, and Rogerio Eduardo Garcia. Challenges to integrate software testing into introductory programming courses. In 2017 IEEE Frontiers in Education Conference (FIE), pages 1–9, 2017a. doi: 10.1109/FIE.2017.8190557. 38
- Lilian Passos Scatalon, Ellen Francine Barbosa, and Rogerio Eduardo Garcia. Challenges to integrate software testing into introductory programming courses. In 2017 IEEE Frontiers in Education Conference (FIE), pages 1–9. Ieee, 2017b. 2

Jeroen J. G. Van Merriënboer and Hein P. M. Krammer. Instructional strategies and tactics for the design of introductory computer programming courses in high school. Instructional Science, 16(3):251–285, September 1987. ISSN 1573-1952. doi: 10.1007/BF00120253. URL <https://doi.org/10.1007/BF00120253>. 4

J.J.G. van Merriënboer and P.A. Kirschner. Ten Steps to Complex Learning: A Systematic Approach to Four-component Instructional Design. Lawrence Erlbaum Associates, 2007. ISBN 978-0-8058-5792-4. URL <https://books.google.nl/books?id=YDWfs7eVJW8C>. 4, 9, 19, 22, 24

APPENDIX A: PROCEDURAL GUIDE

Black-box Testen

Procedure testcases opstellen

VOORAF:

1. Maak het project testgereed (testlibraries, testfolder).
2. Maak een lege implementatie zonder logica voor de methode die je gaat ontwikkelen. De signatuur komt uit de specificatie. Neem de specificatie ook over als documentatie voor de implementatie.
3. Maak een test-class met de naamgeving `<ClassToTest>Test` voor de class waar je testen voor gaat definiëren (kan misschien automatisch met sneltoets afhankelijk van de IDE).

ONTWERP DE TEST:

4. Bedenk voor ieder @subcontract minimaal 1 test.
5. Verzin voor elk van deze test een testwaarde die voldoet aan @requires én definieer hierbij een orakel. Denk ook aan boundaries. Vaak zijn er per subcontract meerdere testcases nodig.

CODEER DE TEST:

6. Schrijf de testmethode:
 - gebruik de @Test annotatie.
 - Indien je een JML:@signals wil testen gebruik je @Test(expected = <Exception>.class).
 - Bepaal de testmethode-signatuur: public void <test-method-name>(). Laat in de <test-method-name> terugkomen:
 - welke methode je test.
 - welke testcase je test.
 - welke Exception of postconditie je verwacht (orakel).
 - Ga door met de body van de test, stappen 7-9.
7. Arrange: Zet de waarden en objecten klaar die je nodig hebt voor de test (uit stap 5).
8. Act: Voer de methode uit. Dit is de methode die je beschreven hebt in de testmethode signatuur.

9. Assert: Controleer of de postconditie overeenkomt met het orakel uit stap 5. Als je een Exception als orakel hebt schrijf je de assertion in de @Test(expected = ...) annotatie.
10. Indien er nog meer tests zijn voor dit subcontract, herhaal vanaf stap 5.
11. Indien er nog meerdere subcontracten te testen zijn, herhaal vanaf stap 4.

MAAK DE IMPLEMENTATIE, DE TESTS ZIJN GECODEERD:

12. Voer de tests uit, je ziet dat er tests falen
13. Maak/wijzig de implementatie stap voor stap ga telkens naar stap 12. Als alle tests slagen is de methode gereed en voldoe je aan de specificaties die in de tests zijn vastgelegd.

BEGRIPPENLIJST:

Preconditie	Situatie waaraan voldaan moet worden voordat de methode uitgevoerd wordt.
Postconditie	Situatie die actief is na uitvoering van de methode.
Orakel	Een verwachte uitkomst van een test die onveranderlijk is.
Inputdomein	Alle mogelijke (combinaties) van inputwaarden.
Equivalentieklassen	Precondities die op eenzelfde wijze een postconditie verkrijgen.
Boundaries	Grenswaarden van het inputdomein.

JML NOTATIES:

@subcontract	Een partitie van pre- met bijbehorende postconditie ¹ .
@requires	Preconditie waaraan voldaan moet worden voor de methode wordt uitgevoerd.
@ensures	Postconditie welke de situatie aangeeft zoals die na uitvoer van de methode geldt.
@signals	Een exceptie die opgeworpen wordt.
@non_null	Een parameter die geen null verwijzing mag hebben.

¹@subcontract behoort niet tot de set van JML annotaties, deze is als alternatief van @also toegevoegd om duidelijker onderscheid tussen partities te kunnen maken.

JUNIT ANNOTATIES:

@BeforeClass	Code die uitgevoerd wordt vóóordat alle testcases in deze test-class gestart worden.
@AfterClass	Code die uitgevoerd wordt nadat alle testcases in deze test-class uitgevoerd zijn.
@Before	Code die telkens vóór iedere testcase uitgevoerd wordt.
@After	Code die telkens na iedere testcase uitgevoerd wordt.
@Test	De daadwerkelijke testcase. Voor iedere test-partitie minimaal 1 testcase.
@Test(expect...)	Een testcase waarin een expected Exception aanwezig is (JML:@signals).

APPENDIX B: LESSON MATERIAL

This appendix shows the lesson material used for the lectures and the practice assignments between lessons. The instructor notes are included beneath the sheets so that can be seen what the focus points are.

Lesson 1 - materials:



Training 9 december Leerdoelen:

- Wat is kwaliteit in code
- Onderscheid kunnen maken in kwaliteitsaspecten onderhoudbaarheid en correctheid
- Kunnen bedenken van zo compleet mogelijke testsuits (verkrijgen van inzicht)
- Lezen en begrijpen van expliciete specificaties in JML
- Het kunnen opstellen van testcases op basis van specificaties in JML, dmv procedural guide

Waarom mislukken vaak softwareprojecten

Ict-fout Belastingdienst kost kabinet 445

20 februari 2019 10:48 | 5



Man
Als je kijkt naar het werk dat ze afleverden dan is de kwaliteit abominabel.

Man2
De code ziet eruit alsof het gemaakt is door beginners die voor het eerst zijn gaan programmeren.

Man3
Ik heb maar een woord daarvoor. Spaghetticode.

in f t w G+

Ceppellini moet ruim 21 miljoen euro betalen aan de Sociale Verzekeringsbank (SVB) vanwege een mislukte automatiseringpro...
Een adviescommissie heeft dat vrijdag bevestigd. heeft staatssecretaris Jette (Sociale Zaken en Werkgelegenheid) zaterdag aan de Tweede Kamer laten v...
Ceppellini heeft maandag in een *verklaring* aan investeerders laten weten te...
in besprek te gaan tegen de uitkomst.

int.
Zo heet dat ook.

systeemfout aan het licht of vermogensgrens werd bijvoorbeeld doordat het moest wel gebeuren.

De SVB stopte in 2014 met de ontwikkeling van het systeem voor betalingen waaraan of jaren werd gewerkt. De uitvoeringsomvang had toen 43,7 miljoen euro gestoken in het project.

GERELATEERDE ARTIKELN
UNY en SVB nu burgers laten bij over afgehuurd
VVD heeft geen ICT-mislukking bij

Zo heet dat ook een onontwerpbare kluisen van, van code.

Voicé-over

Programmers

De SVB is uitvoerder van een aantal regelingen in de sociale zekerheid en

Zembla onderzoekt waarom grote ICT projecten bij de overheid zo vaak mislukken en wie daar baat bij hebben.

Kwaliteit



- Wanneer is je code goed?
 - Als deze compileert?
 - Als je een test doet en de verwachting klopt?
 - Als je een overzichtelijke structuur van je code hebt?
 - Als ...

www.menti.com

Programmeren 2

19 augustus 2021

13

Laat studenten zaken opsommen in het maken van code waaraan je kwaliteit kunt bepalen. Gebruik Mentimeter wordcloud. De genoemde zaken gaan we daarna uitschrijven in de volgende sheet.

Kwaliteit

Interne kwaliteit (onderhoudbaarheid)

Externe kwaliteit (correctheid)

Structuur
Overzichtelijk
Leesbaar voor anderen
Naamgeving variabelen/methoden

- Doet de code wat hij moet doen
- Juiste exceptions bij onjuiste input
- Voor alle situaties juiste werking

Toevoegen nav input studenten. Zorg iniedersgeval voor:

- Goede naamgeving variabelen (intern)
- Werking zoals verwacht (extern)
- Begrijpbare code, goed leesbaar (intern)
- Goede exceptions bij onjuiste input (extern)

Focus voor nu: EXTERNE KWALITEIT!

Oefening (10 min.)

Download calculateAge.class van Blackboard:

- Analyseer de correctheid van calculateAge.class
 - Opstarten met: `java calculateAge <dd-mm-yyyy>`
 - Laat weten, werkt de code goed/niet goed
 - In welke situatie wel, of niet?
 - Wat heb je allemaal getest (maak een lijstje)

**Hoe wist je wat de methode moest doen?
Aannames?**

Wat heb je getest:

- verjaardag moet nog komen dit jaar
- verjaardag is al geweest dit jaar
- wat als je verjaardag in de toekomst ligt? Kan dit eigenlijk wel, of moet dit een Exception opleveren?

Welke waarden heb je gebruikt? Je eigen geboortedatum? Kan je dit gebruiken in geautomatiseerde tests?

Maar nog belangrijker: Hoe kan je correctheid analyseren als niet gespecificeerd is wat de methode moet doen! Stel dat CalculateAge de leeftijd in maanden zou moeten berekenen. Nu baseren we de specificaties op aannames!

Specificaties - omschrijving



- Wat zijn de specificaties van de methode:
public static int daysUntilNextBirthday(LocalDate dateOfBirth)
- De beoogde werking van deze methode is:
Berekent het totaal aantal dagen tot de volgende verjaardag van dateOfBirth.

Programmeren 2 19 augustus 2021 6

Discussie: hoe weten we wat de methode doet? Naamgeving van de methode, maar hoe expliciet onthult deze de werking.

Specificaties - preconditionie



- Wat zijn de specificaties van de methode:
public static int daysUntilNextBirthday(LocalDate dateOfBirth)
- Waar moet de parameter 'dateOfBirth' aan voldoen?
dateOfBirth moet een referentie naar een LocalDate hebben
dateOfBirth moet geldig zijn (is verantwoordelijkheid LocalDate)
dateOfBirth kan niet in de toekomst liggen (anders Exception)

Programmeren 2 19 augustus 2021 7

Discussie: laat studenten verschillende situaties voor dateOfBirth onderscheiden. Wat is wel toegestaan, en wat niet? Mag een geboortedatum in de toekomst liggen? Uitgangspunt: LocalDate is altijd geldig, het is dus niet mogelijk een datum van 31 november te zetten. Dit valt dus buiten de verantwoordelijkheid van daysUntilNextBirthday.

Specificaties - postconditie

- Wat zijn de specificaties van de methode:
`public static int daysUntilNextBirthday(LocalDate dateOfBirth)`
- Wat valt er te zeggen over de output?

**result is het aantal dagen tot de volgende verjaardag
is 0 geldig? -> hint: *NextBirthday*
result kan 1 tot 365 zijn**

**Als bij het tellen een schrikkelag gepasseerd wordt
kan result ook 365 zijn.**

Maar.. als iemand jarig op 29 februari, hoeveel dagen?

Programmeren 2

19 augustus 2021

18

Discussie: Het kan nooit 0 dagen duren tot je volgende verjaardag. Taalkundig verhult de methode naam dat het aantal dagen minimaal 1 is en maximaal 1 jaar duurt als je precies deze methode uitvoert op je verjaardag. Maar is deze aanname juist? Hoe lang duurt 1 jaar... 365 dagen? Als je een schrikkelag passeert zou je volgende verjaardag best eens 366 kunnen zijn. Maar, als je op een schrikkelag geboren bent zou je volgende verjaardag best meer als 1000 dagen kunnen zijn.

Specificaties

Waarom:

- Expliciet maken van aannames
- Ontwerpbeslissingen vastleggen

Hoe:

- Natuurlijke taal (gemakkelijk anders te interpreteren)
- JML (Java Modeling Language)
 - Pre- en Postcondities
 - Exceptions

We leggen dus een contract vast waar gebruikers van de methode, en de methode zelf, zich aan moeten houden.

Programmeren 2

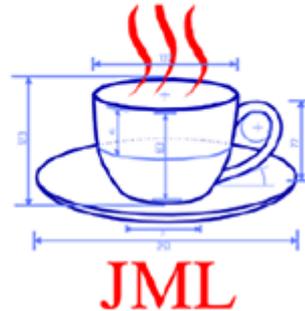
19 augustus 2021

19

Een probleemomschrijving in natuurlijke taal kunnen we expliciteren naar specificaties in JML (dit expliciteren wordt ook wel 'design by contract' genoemd). Feitelijk leggen we een contract vast waarbij de aanroeper van de methode zich moet houden aan de pre-condities en de methode zich vervolgens houdt aan de postcondities.

JML – Java Modeling Language

- Formeler dan JavaDoc, maakt specificaties expliciet:
 - Pre-condities (@requires)
 - Post-condities (@ensures)
 - Exceptions (@signals)



UITLEGGEN:

Waarom formeler: minder kans op misinterpretatie door expliciet, pre-, post-condities, exceptions zo eenduidig mogelijk interpreteerbaar te formuleren. Pre-condities is alles waaraan voldaan moet worden voordat de methode uitgevoerd wordt Post-condities, wat 'het' uitvoeren van de methode tot gevolg heeft (hoeft niet een returnwaarde te zijn, maar ook verandering van state in een object bijv.) Exceptions, indien de bewerking een uitzondering veroorzaakt **Doel is nu niet het opstellen van de specificaties, maar het begrijpen van de notatie en interpreteren van de specificaties.**

Omschrijving in JML

Berekent het totaal aantal dagen tot de volgende verjaardag van dateOfBirth.

```
/**  
 * @desc Calculates the amount of days until the next birthday of dateOfBirth.  
 */  
public static int daysUntilNextBirthday(LocalDate dateOfBirth);
```

Discussie: hoe weten we wat de methode doet? Naamgeving van de methode, maar hoe expliciet onthult deze de werking. Geeft de description voldoende informatie om zonder aannames te maken de implementatie te maken? Nee, we hebben meer informatie nodig.

Preconditie in JML

- dateOfBirth moet een referentie naar een LocalDate hebben
- dateOfBirth moet geldig zijn (is verantwoordelijkheid LocalDate)
- dateOfBirth kan niet in de toekomst liggen (anders Exception)

```
/**
 * @desc Calculates the amount of days until the next birthday of dateOfBirth.
 *
 * @requires dateOfBirth != null && dateOfBirth <= LocalDate.now();
 * @signals (IllegalArgumentException) dateOfBirth > LocalDate.now();
 * @signals (NullPointerException) dateOfBirth == null;
 */
public static int daysUntilNextBirthday(LocalDate dateOfBirth);
```

Het komt vaak voor dat referenties niet null mogen zijn, daarom is er een eenvoudige verkorte notatie voor: non_null

Een geïnstantieerde LocalDate is altijd geldig, een datum van 31 november is dus niet mogelijk. Dit zegt echter niets over de geldigheid van de LocalDate voor onze methode! Als de methode aangeroepen wordt MOET aan de preconditione voldaan worden. Indien niet voldaan wordt aan de preconditione wordt een 'exception' opgegooid. Exceptions worden expliciet gespecificeerd en moeten dus ook als zodanig geïmplementeerd worden.

Preconditie in JML

- dateOfBirth moet een referentie naar een LocalDate hebben
- dateOfBirth moet geldig zijn (is verantwoordelijkheid LocalDate)
- dateOfBirth kan niet in de toekomst liggen (anders Exception)

```
/**
 * @desc Calculates the amount of days until the next birthday of dateOfBirth.
 *
 * @requires dateOfBirth <= LocalDate.now();
 * @signals (IllegalArgumentException) dateOfBirth > LocalDate.now();
 * @signals (NullPointerException) dateOfBirth == null;
 */
public static int daysUntilNextBirthday(/*@non_null */ LocalDate dateOfBirth);
```

Postconditie in JML

- result kan 1 tot en met 364 zijn
- Als bij het tellen een schrikkel dag gepasseerd wordt kan result 365 zijn.

```
/**
 * @desc Calculates the amount of days until the next birthday of dateOfBirth.
 *
 * @requires dateOfBirth <= LocalDate.now();
 * @signals (IllegalArgumentException) dateOfBirth > LocalDate.now();
 * @signals (NullPointerException) dateOfBirth == null
 * @ensures \result = #days until birthday (be careful when born on or have to count feb 29th);
 */
public static int daysUntilNextBirthday( /*@ non null @*/ LocalDate dateOfBirth);
```

Wanneer test je nu goed?

- Equivalence classes (nb: dit heeft niets met Java class te maken)
 - Ingedeeld in partities met gelijkvormige werking
- Boundaries
 - Randgevallen

**Wat ga je testen? Verzin voldoende testcases.
Oefening 10 min.**

**Om het voorbeeld niet meteen te complex te maken
vergeten we even dat er ook schrikkeljaren
bestaan.**

Hoe gaan we nu testen adhv de specificaties. Laat studenten 'testregels' verzinnen. Welke verschillende situaties kun je onderscheiden:

- dateOfBirth > now
- dateOfBirth == null
- Al jarig geweest dit jaar, Nog jarig dit jaar, Vandaag jarig
- Er is geen 29 februari aanwezig tot de volgende verjaardag
- Er is wel 29 februari aanwezig tot volgende verjaardag
- Geboren op 29 februari

JML – Subcontracten

- Expliciet partities onderscheiden:

```
@subcontract: case {  
  @requires ...  
  @ensures ...  
}
```



UITLEGGEN: Expliciet aangegeven subcontracten dienen getest te worden. Meerdere subcontracten (partities) zouden samen het inputdomein moeten afdekken. **Doel is nu niet het opstellen van de specificaties, maar het begrijpen van de notatie en interpreteren van de specificaties.**

Ik ben nog niet geboren

```
/**  
 * @desc Calculates the amount of days until the next birthday of dateOfBirth.  
 *  
 * @subcontract: not born yet {  
 *   @requires dateOfBirth > LocalDate.now();  
 *   @signals (IllegalArgumentException) dateOfBirth > LocalDate.now();  
 * }  
 *  
 * ...  
 *  
 public static int daysUntilNextBirthday(/**@ non null */ LocalDate dateOfBirth);
```

Ongeldige waarde, moet een exception opgooien.

dateOfBirth == null

```
/**
 * @desc Calculates the amount of days until the next birthday of dateOfBirth.
 *
 * @subcontract: null dateOfBirth {
 *   @requires dateOfBirth == null;
 *   @signals (NullPointerException) dateOfBirth == null;
 * }
 *
 * ...
 *
 public static int daysUntilNextBirthday(/**@ non null */ LocalDate dateOfBirth);
```

Ongeldige waarde, moet een exception opgooien.

Ik ben nog niet jarig geweest dit jaar

```
/**
 * @desc Calculates the amount of days until the next birthday of dateOfBirth.
 *
 * @subcontract: birthday is yet to come this year {
 *   @requires LocalDate.now().getMonthValue() < dateOfBirth.getMonthValue() ||
 *     LocalDate.now().getMonthValue() == dateOfBirth.getMonthValue() &&
 *     LocalDate.now().getDayOfMonth() < dateOfBirth.getDayOfMonth();
 *   @ensures \result = #days until (not including) next birthday && 1 <= \result < 365;
 * }
 *
 * ...
 *
 public static int daysUntilNextBirthday(/**@ non null */ LocalDate dateOfBirth);
```

Ik ben jarig vandaag

```

/**
 * @desc Calculates the amount of days until the next birthday of dateOfBirth.
 *
 * @subcontract: today is my birthday {
 * @requires dateOfBirth.getMonthValue() == LocalDate.now().getMonthValue() &&
 *           dateOfBirth.getDayOfMonth() == LocalDate.now().getDayOfMonth();
 * @ensures \result = 365;
 * }
 * ...
 *
 public static int daysUntilNextBirthday(/*@ non null */ LocalDate dateOfBirth);

```

Uiteraard geldt dit alleen in de situatie dat er geen schrikkeljaren bestaan!
 Discussie:

- Welke situaties kunnen we onderscheiden in geval dat er 29 februari gepasseerd wordt.
- Wat als ik op 29 februari jarig ben!

Ik ben al jarig geweest dit jaar

```

/**
 * @desc Calculates the amount of days until the next birthday of dateOfBirth.
 *
 * @subcontract: birthday has already passed this year {
 * @requires dateOfBirth.getMonthValue() < LocalDate.now().getMonthValue() ||
 *           dateOfBirth.getMonthValue() == LocalDate.now().getMonthValue() &&
 *           dateOfBirth.getDayOfMonth() < LocalDate.now().getDayOfMonth();
 * @ensures \result = #days until (not including) next birthday && 1 <= \result < 365;
 * }
 *
 public static int daysUntilNextBirthday(/*@ non_null */ LocalDate dateOfBirth);

```

Hoe ga je dit testen

- Download DaysUntilMyNextBirthday.class van blackboard.

**Hoe ga je testen? Voer voldoende testcases uit.
Oefening 10 min.**

Hoe ga je dit testen

- Welke precondities heb je verzonnen?
- Hoe wist je of de test geslaagd was.

Het orakel

- Wat is een orakel
 - Een orakel geeft voorspelling van de juiste postconditie voor een preconditie.
 - Een orakel geldt altijd! (Morgen dus ook ☺)



Procedural guide black-box testen

1. Maak voor ieder @subcontract minimaal 1 testcase (denk aan randgevallen).
2. Verzin per testcase een precondition die voldoet aan @ensure én definieer hierbij een orakel.
3. Voer de methode uit met de precondition uit stap 2.
4. Controleer of de postconditie overeenkomt met het orakel uit stap 2.
5. Indien stap 4 positief is, is de test geslaagd.

Procedural guide black-box testen

```
* @subcontract: not born yet (  
*   @requires dateOfBirth > LocalDate.now();  
*   @signals (IllegalArgumentException) dateOfBirth > LocalDate.now();  
* )
```

Testcase	Not born yet
Preconditie	
Orakel	
Postconditie of Exception	

Procedural guide black-box testen

```
* @subcontract: not born yet (  
*   @requires dateOfBirth > LocalDate.now();  
*   @signals (IllegalArgumentException) dateOfBirth > LocalDate.now();  
* )
```

Testcase	Not born yet
Preconditie	dateOfBirth = LocalDate.now().plusDays(10)
Orakel	
Postconditie of Exception	

Waarom is dateOfBirth = 25-12-2020 fout hier?

Procedural guide black-box testen

```
* @subcontract: not born yet (  
*   @requires dateOfBirth > LocalDate.now();  
*   @signals (IllegalArgumentException) dateOfBirth > LocalDate.now();  
* )
```

Testcase	Not born yet
Preconditie	dateOfBirth = LocalDate.now().plusDays(10)
Orakel	IllegalArgumentException
Postconditie of Exception	

Procedural guide black-box testen

```
* @subcontract: not born yet (  
*   @requires dateOfBirth > LocalDate.now();  
*   @signals (IllegalArgumentException) dateOfBirth > LocalDate.now();  
* )
```

Testcase	Not born yet
Preconditie	dateOfBirth = LocalDate.now().plusDays(10)
Orakel	IllegalArgumentException
Postconditie of Exception	IllegalArgumentException

Oefenen in de studiegroepen



- **Testcases opstellen (opdracht van Blackboard)**
 - Gebruik de invulsjablonen in de opdracht
 - Download de classfiles van Blackboard
 - Voer de testen. Voorbeeld:
java getAnnualBonusPercentage 25
- **Als je meer uitdaging zoekt in het testen:**
 - Opdracht 1 testcases verzinnen rekening houdend met schnikkeljaren.
- **Daarna verder werken aan MOOC opdrachten**
- **Databasekoppeling werkend krijgen (handleiding bb)**
Test dit op tijd want dit heb je nodig voor de groepsopdracht.

Lesson 2 - materials:



avans hogeschool

PROGRAMMEREN 2
Training 8

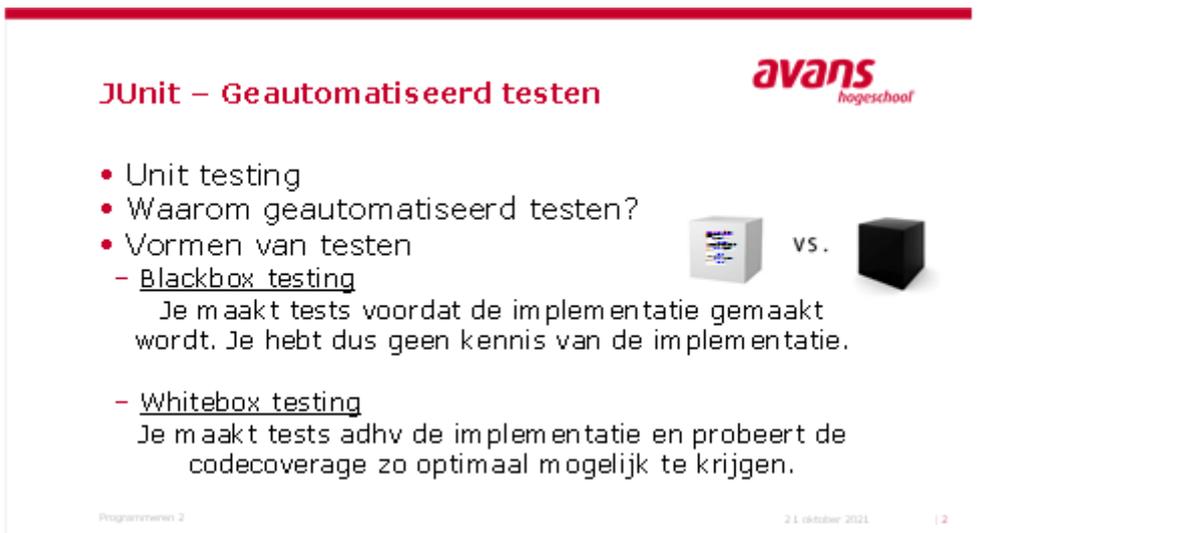
Informatica Breda

Programmeren 2 21 oktober 2021

Training 17 december

LEERDOELEN:

- Het nut begrijpen van het opstellen van geautomatiseerde tests.
 - Het schrijven van testcode in Junit -> Arrange, Act, Assert.
 - Junit essentials: Setup, Teardown, TestSuites.
 - Stap 2 uit de procedural guide: testcases uitschrijven in code.
-



avans hogeschool

JUnit – Geautomatiseerd testen

- Unit testing
- Waarom geautomatiseerd testen?
- Vormen van testen
 - Blackbox testing
Je maakt tests voordat de implementatie gemaakt wordt. Je hebt dus geen kennis van de implementatie.
 - Whitebox testing
Je maakt tests adhv de implementatie en probeert de codecoverage zo optimaal mogelijk te krijgen.

Programmeren 2 21 oktober 2021 12

Handmatig testen kost veel tijd. Automatiseren geeft ook de mogelijkheid tot regressietesten. Black-box tests worden gebruikt bij TDD. TDD is erg 'hip', XP is erop gebaseerd. Stappen zijn: eerst testen schrijven, daarna de implementatie opbouwen totdat de testen slagen. Hierna refactoren zodat de code netjes en onderhoudbaar is. Wij gaan in de module programmeren 2 deze methodiek

toepassen. De nadruk blijft nog even liggen op externe kwaliteit (correctheid) en niet het refactoren om de interne kwaliteit te verhogen (onderhoudbaarheid). Bij 'Whitebox' tests gaan we proberen alle codepaden in de implementatie te bewandelen. Risico is groot dat je fouten in de implementatie mee overneemt in de tests omdat je het orakel gaat baseren op de implementatie. Hierdoor bestaat er een grotere kans afbreuk te doen aan externe kwaliteit. Blackbox tests hoeven Whitebox tests niet uit te sluiten! Echter beperken we ons in deze module tot Blackbox tests.

Automatisch testen



We gebruiken hiervoor de JUnit library:

- Tool/library voor het uitvoeren van Java (unit)testcode
- 'Automatische' beoordeling van testresultaten
- **Assertions** (beweringen) voor het testen op orakels
- **setUp** en **tearDown** voor 'opzetten en opruimen van objecten/vars'
- **TestSuites** voor het organiseren van TestCases
- Grafische en tekstuele test runners



Programmeren 2

We gaan in de les alleen unittests maken met behulp van Assertions. `setUp` en `tearDown` zijn methoden welke na afloop en voor de test uitgevoerd worden. Soms kun je codeduplicatie hiermee voorkomen in testcode. Hoe dit werkt kun je zelf uitzoeken in de JUnit documentatie. Met TestSuites kun je testen organiseren, zie Documentatie

Een voorbeeld - Asiris.



De productowner van Asiris (een nieuw cijferregistratiesysteem voor scholen) wil functionaliteit voor het automatisch omzetten van cijfers naar woordbeoordelingen toevoegen.

Eisen:

- 1 tot 5.5: onvoldoende
- 5.5 tot 7: voldoende
- 7 tot 8: ruim voldoende
- 8 tot 9: goed
- 9 tot 10: uitmuntend

Programmeren 2 21 oktober 2021 | 4

De eisen in natuurlijke taal lijken duidelijk, ze zijn echter niet 100% correct. Kunnen studenten dit vinden? Er worden zeker aannames vereist, als voorbeeld: als onvoldoende behoort tot 5.5 en niet inclusief) wat voor woordbeoordeling krijg je dan bij een 10? Wat als de cijfer-beoordelingen buiten bereik vallen? Daarom specificatie in JML. (herhaling)

We hebben een project! 🤖

Test driven development

Doel:

- Gestructureerde aanpak.
- Gebruik maken van expliciete specificaties.
- 'Garanties' geven op correctheid.
- Regressietests:
zonder moeite blijven we kwaliteit bewaken.

Hoe:

- Een stappenplan! -> zie Blackboard.

Het gebruik maken van expliciet opgestelde specificaties zal in de praktijk weinig voorkomen. We doen dit nu om studenten bewust te laten worden om precieze definities te hanteren en om niet te gemakkelijk aannames te doen. Op het moment dat bewustwording is aangeleerd is het eenvoudiger de stap te maken naar specificaties opstellen. Het stappenplan is een trainingsmiddel om tests schrijven te leren automatiseren. Hierbij ook bewust te zijn van equivalentieklassen en grenswaarden. Laat de procedure zien en doorloop deze voordat we verdergaan.

Een voorbeeld - Asiris.

```
/**
 * @desc returns a dutch word-rated grade based on a numeric grade.
 *
 * @subcontract inadequate {
 *   @requires 1 <= grade < 5,5;
 *   @ensures \result = "onvoldoende";
 * }
 */
public static String getWordRating(double grade);

Stap 2: Maak een implementatie die nog niets doet

public static String getWordRating(double grade) {
    return null;
}
```

We gaan blackbox tests maken en we gaan deze tests toepassen zoals dit ook bij TDD gebeurt. De specificatie kunnen we gewoon boven de implementatie plakken, zodoende hebben we meteen goede documentatie. De specificatie is in dit voorbeeld nog niet volledig, op blackboard is de volledige specificatie te vinden van deze casus. Laat deze in de les even zien.

Het aanmaken van een test-class

```
public static String getWordRating(double grade) {  
    return null;  
}
```

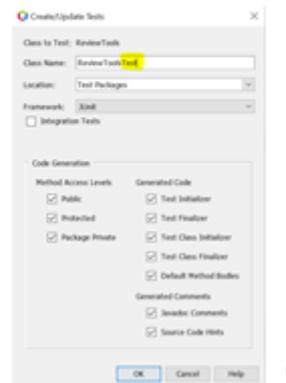
Stap 3: Maak een test(class). (Netbeans:
<CTRL><ALT>T)

Je kunt automatisch templatecode laten genereren.

Javaconventie: Classname voor testen wordt de
Classname+ Test

In je projectstructuur worden de tests gescheiden van je
implementatieproject opgeslagen. Waarom zou dit zijn?

Programmeren 2



We gaan black-box tests maken en we gaan deze tests toepassen zoals dit ook bij TDD gebeurt. Ga op de methode staan en druk <CTRL><ALT>T. Laat studenten bedenken waarom je tests gescheiden van de implementatie wil opslaan. Je wil de tests in de deploymentomgeving niet mee distribueren.

JUnit testannotaties:

Testannotaties:

- **@BeforeClass** -> code die uitgevoerd wordt voordat alle testcases in deze test-dass gestart worden.
- **@AfterClass** -> code die uitgevoerd wordt nadat alle testcases in deze test-dass uitgevoerd zijn.
- **@Before** -> code die telkens vóór iedere testcase uitgevoerd wordt.
- **@After** -> code die telkens na iedere testcase uitgevoerd wordt.
- **@Test** -> De daadwerkelijke testcase. Voor iedere test-partitie minimaal 1 testcase.

```
public class ReviewToolsTest {  
    public ReviewToolsTest() {  
    }  
    @BeforeClass  
    public static void setUpClass() {  
    }  
    @AfterClass  
    public static void tearDownClass() {  
    }  
    @Before  
    public void setUp() {  
    }  
    @After  
    public void tearDown() {  
    }  
    @Test  
    public void testScoreMethod() {  
        // TODO review the generated test code and remove the  
        // default call to fail.  
        fail("The test case is a prototype.");  
    }  
}
```

Programmeren 2

21 oktober 2021

Uitleg verschillende annotaties. Alle annotaties buiten @Test laten we nog even buiten beschouwing, hiermee kunnen we codeduplicatie voorkomen maar we moeten wel goed opletten wat we hiermee doen! Let op dat je bijvoorbeeld geen afhankelijkheden tussen de verschillende testen gaat creëren. Focus nu op @Test -> definitie van de testcase.

Een voorbeeld - Asiris.

```
* @subcontract inadequate (
*   @requires 1 <= grade < 5,5;
*   @ensures \result = "onvoldoende"; ) */
public static String getWordRating(double grade);
```

Stap 4+5: Ontwerp de test (zie ook procedure vorige les)

Testcase	Onvoldoende
Preconditie	4,0
Oracle	equal("onvoldoende")
Postconditie of Exception	

Programmen 2

21 oktober 2021

| 9

We gaan blackbox tests maken en we gaan deze tests toepassen zoals dit ook bij TDD gebeurt. De specificatie kunnen we gewoon boven de implementatie plakken, zodoende hebben we meteen goede documentatie.

De test-methode

- Net zoals bij de implementatie willen we dat een test een 'alleszeggende' naamgeving heeft.

- Stap 6: formule:

```
Methodenaam = "test" +
  <methodenaamDieGetestWordt> +
  Requires +
  <preconditie> +
  Ensures/Signals (naar keuze) +
  <postcoditie>/<exception> (afhankelijk Ensures/Signals)
```

Programmen 2

21 oktober 2021

| 10

- Een test gaan we uitvoeren als 'drietraps-raket'
 - **Stap 7:** Arrange: klaarzetten
Objecten aanmaken, data(-sets) voorbereiden. Opbouwen van de pre-conditie.
 - **Stap 8:** Act: uitvoeren van de 'methode onder test'
Aanroep van de methode en afvangen van het resultaat.
 - **Stap 9:** Assert: Controleren van de postconditie -> uitkomst vergelijken met het orakel.

Programmen 2

21 oktober 2021

| 11

Deze drie stappen gaan we precies zo en in deze volgorde uitvoeren per testsituatie. Leer jezelf aan deze stappen steeds als documentatie in je testcode te laten terugkomen!

Een voorbeeld - Asiris.

Stap 6+7+8+9: Codeer de test

```
/**
 * @subcontract inadequate {
 * @requires 1 <= grade < 5,5;
 * @ensures \result = "onvoldoende"; }
 */
@Test
public void testGetWordRatingRequires4EnsuresOnvoldoende() {
    //Arrange
    double grade = 4.0;

    //Act
    String result = ReviewTools.getWordRating(grade);

    //Assert
    assertEquals("onvoldoende", result);
}
```

Programmen 2

21 oktober 2021

| 12

Testcase	onvoldoende
Preconditie	4,0
Orakel	equals("onvoldoende")
Postconditie of Exception	

We gaan blackbox tests maken en we gaan deze tests toepassen zoals dit ook bij TDD gebeurt. Het subcontract uit specificatie kunnen we gewoon boven de test plakken, zodoende hebben we meteen goede documentatie. Hetzelfde doen we voor de 1,0 waarde.

Stap 10: meer? Boundaries?

```
* @subcontract inadequate {
* @requires 1 <= grade < 5,5;
* @ensures \result = "onvoldoende"; } */
public static String getWordRating(double grade);
```

Stap 4+5: Ontwerp de test (zie ook procedure vorige les)

Testcase	onvoldoende bij onder 1,0
Preconditie	1,0
Orakeel	equalTo("onvoldoende")
Postconditie of Exception	

Programmen 2

21 oktober 2021

| 13

We gaan blackbox tests maken en we gaan deze tests toepassen zoals dit ook bij TDD gebeurt. De specificatie kunnen we gewoon boven de implementatie plakken, zodoende hebben we meteen goede documentatie.

Een voorbeeld - Asiris.

Stap 6+7+8+9: Codeer de test

```
/**
* @subcontract inadequate {
* @requires 1 <= grade < 5,5;
* @ensures \result = "onvoldoende"; }
*/
@Test
public void testGetWordRatingRequires4Ensures0nVoldoende() {
    //&rrange
    double grade = 1.0;

    //&act
    String result = ReviewTools.getWordRating(grade);

    //&assert
    assertEquals("onvoldoende", result);
}
```

Programmen 2

21 oktober 2021

| 14

Step 10: meer? Boundaries?

```
* @subcontract inadequate (
* @requires 1 <= grade < 5,5;
* @ensures \result = "onvoldoende"; ) */
public static String getWordRating(double grade);
```

Stap 4+5: Ontwerp de test (zie ook procedure vorige les)

Testcase	InadequateBoundaryHigh
Preconditie	5,4999
Orkkel	equals("onvoldoende")
Postconditie of Exception	

Programmeren 2

21 oktober 2021

| 15

We gaan blackbox tests maken en we gaan deze tests toepassen zoals dit ook bij TDD gebeurt. De specificatie kunnen we gewoon boven de implementatie plakken, zodoende hebben we meteen goede documentatie.

Een voorbeeld - Asiris.

Stap 6+7+8+9: Codeer de test

```
/**
 * @subcontract inadequate (
 * @requires 1 <= grade < 5,5;
 * @ensures \result = "onvoldoende"; )
 */
@Test
public void testGetWordRatingRequires5point4999EnsuresOnvoldoende() {
    //&rrange
    double grade = 5.4999;

    //&act
    String result = ReviewTools.getWordRating(grade);

    //&assert
    assertEquals("onvoldoende", result);
}
```

Programmeren 2

21 oktober 2021

| 16

Een voorbeeld - Asiris.

Stap 10: Er zijn nog meer subcontracten te behandelen

```
* @subcontract out of range grade {  
*   @requires grade < 1 || grade > 10;  
*   @signals (IllegalArgumentException) grade < 1 || grade > 10;  
* }
```

Voor Exceptions zien de tests er iets anders uit. De **assert** wordt in de annotatie gedaan:

```
@Test(expected = IllegalArgumentException.class)  
public void testGetWordRatingRequires0point9SignalIllegalArgumentException()
```

Let op, in Junit 5 wordt dit op een andere manier gedaan.

We gaan blackbox tests maken en we gaan deze tests toepassen zoals dit ook bij TDD gebeurt. De specificatie kunnen we gewoon boven de implementatie plakken, zodoende hebben we meteen goede documentatie.

Een voorbeeld - Asiris.

```
* @subcontract out of range grade {  
*   @requires grade < 1 || grade > 10;  
*   @signals (IllegalArgumentException) grade < 1 || grade > 10;  
* }
```

Stap 4+5: Ontwerp de test. (zie ook procedure vorige les)

Testcase	OutOfRangeLow
Preconditie	0,9
Orakeel	throw IllegalArgumentException
Postconditie of Exception	

Testcase	OutOfRangeExceptionHigh
Preconditie	10,1
Orakeel	throw IllegalArgumentException
Postconditie of Exception	

We gaan blackbox tests maken en we gaan deze tests toepassen zoals dit ook bij TDD gebeurt. De specificatie kunnen we gewoon boven de implementatie plakken, zodoende hebben we meteen goede documentatie.

Een voorbeeld - Asiris.

Stap 6+7+8+9: Codeer de test

```
/**
 * @subcontract out of range grade {
 *   @requires grade < 1 || grade > 10;
 *   @signals (IllegalArgumentException) grade < 1 || grade > 10;
 * }
 */
@Test(expected = IllegalArgumentException.class)
public void testGetWordRatingRequires0point9signalsIllegalArgumentException() {
    //Arrange
    double grade = 0.9;

    //Act
    String result = ReviewTools.getWordRating(grade);
}

```

Programmeer 2 21 oktober 2021 | 19

Testcase	OutOfRangeLow
Preconditie	0,9
Orakel	throw IllegalArgumentException
Postconditie of Exception	

We gaan blackbox tests maken en we gaan deze tests toepassen zoals dit ook bij TDD gebeurt. De specificatie kunnen we gewoon boven de test plakken, zodoende hebben we meteen goede documentatie. Hetzelfde doen we voor de 10,1 waarde.

Een voorbeeld - Asiris.



Stap 10: Herhaal voor alle subcontracten

Stap 11: Run de tests

The screenshot shows a 'Test Results' window for 'ReviewToolsTest'. At the top, it says 'Tests passed: 0,00 %'. Below that, it states 'No test passed, 13 tests failed, (0,123 s)'. A list of 13 failed tests is shown, each with a red 'X' icon and a failure message. The messages are: 'Failed: Expected exception: java.lang.IllegalArgumentException', 'Failed: expected: <onvoldoende> but was: <null>', 'Failed: expected: <uitmuntend> but was: <null>', 'Failed: expected: <onvoldoende> but was: <null>', 'Failed: expected: <goed> but was: <null>', 'Failed: expected: <ruim voldoende> but was: <null>', 'Failed: expected: <voldoende> but was: <null>', 'Failed: expected: <ruim voldoende> but was: <null>', 'Failed: expected: <goed> but was: <null>', 'Failed: expected: <voldoende> but was: <null>', 'Failed: expected: <uitmuntend> but was: <null>', 'Failed: expected: <onvoldoende> but was: <null>', and 'Failed: Expected exception: java.lang.IllegalArgumentException'.

Programmeer 2 21 oktober 2021 | 20

Na het definiëren van de tests zien we dat de tests falen, logisch want de implementatie moet nog gemaakt worden.

Een voorbeeld - Asiris.

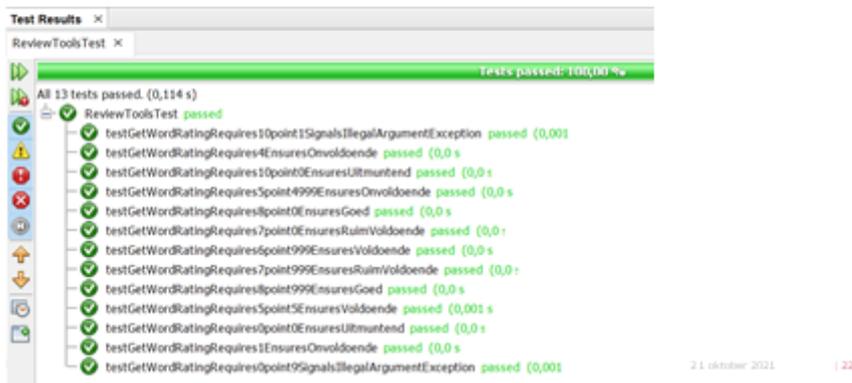
Stap 12: Maak de implementatie, eventueel stap voor stap -> Stap 11.

```
public static String getWordRating(double grade) {  
    if (grade < 1) {  
        throw new IllegalArgumentException("grade out of bounds");  
    } else if (grade < 5.5) {  
        return "onvoldoende";  
    } else if (grade < 7.0) {  
        return "voldoende";  
    } else if (grade < 8.0) {  
        return "ruim voldoende";  
    } else if (grade < 9.0) {  
        return "goed";  
    } else if (grade <= 10.0) {  
        return "uitmuntend";  
    }  
    throw new IllegalArgumentException("grade out of bounds");  
}
```

Na het definiëren van de tests zien we dat de tests falen, logisch want de implementatie moet nog gemaakt worden. De implementatie kun je stap voor stap maken en aan het resultaat van de tests zie je je voortgang.

Een voorbeeld - Asiris.

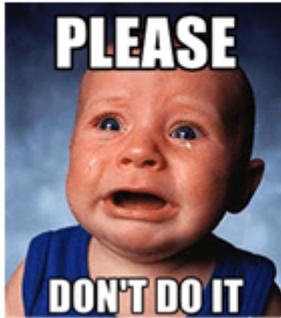
Stap 11: Run de tests



The screenshot shows a 'Test Results' window for 'ReviewToolsTest'. At the top, it indicates 'Tests passed: 100,00 %'. Below this, a summary line states 'All 13 tests passed. (0,114 s)'. A list of 13 individual test cases follows, each with a green checkmark icon and the text 'passed (0,001 s)'. The test cases include:

- testGetWordRatingRequires10point1SignalsIllegalArgumentException
- testGetWordRatingRequires4EnsuresOnvoldoende
- testGetWordRatingRequires10point0EnsuresUitmuntend
- testGetWordRatingRequires3point999EnsuresOnvoldoende
- testGetWordRatingRequires3point0EnsuresGoed
- testGetWordRatingRequires7point0EnsuresRuimVoldoende
- testGetWordRatingRequires5point999EnsuresVoldoende
- testGetWordRatingRequires7point999EnsuresRuimVoldoende
- testGetWordRatingRequires3point999EnsuresGoed
- testGetWordRatingRequires3point5EnsuresVoldoende
- testGetWordRatingRequires0point0EnsuresUitmuntend
- testGetWordRatingRequires1EnsuresOnvoldoende
- testGetWordRatingRequires0point9SignalsIllegalArgumentException

Als alle tests slagen is je implementatie gereed.



Valkuil:

'Het voelt alsof de tests allemaal hetzelfde zijn. Ik schrijf wel één enkele test met een lus van 0 tot 11 waarmee ik alle subcontracten efficiënt kan testen!'

Waarom willen we dit niet?

Programmeren 2

21 oktober 2021

| 23

Laat studenten bedenken wat er mis gaat als je op deze manier tests gaat vereenvoudigen. -> Je introduceert complexiteit en gaat de voorziene implementatie integreren in de test. Dit vergroot niet alleen het risico op onjuiste tests. Tijdens het maken van de implementatie is de verleiding heel groot om logica gebruikt in de test over te gaan nemen in de implementatie. Op deze manier slagen de tests altijd, maar wat kun je dan zeggen over de externe kwaliteit? Ook geven losse tests meer gericht aan wat er misgaat in de implementatie.

Milestone 6

**Bonuspunt voor het programmeergedeelte opdracht
(enkel voor studenten-koppels die de opdracht compleet voldaan hebben)**

**Think-aloud pair-programming
inclusief schermopname en audio.**

Belangrijk:

- Een pair/koppel zijn 2 studenten (een groep heeft 2 koppels)
(Lukt dit niet, neem contact op met je docent)
- Inleveren per koppel: Testcode, Implementatie en Video-opname
- Gedetailleerde instructies staan op blackboard
- **Deadline: 10 januari 2021 23:59**

Programmeren 2

21 oktober 2021

| 24

Details staan in de opdrachtomschrijving.

Oefenen in de studiegroepen

- Voordat je de Milestone gaat uitwerken, eerst oefenen!
- **Testcases opstellen (opdracht van vorige week)**
 - Gebruik de ingevulde sjablonen van de opdracht van afgelopen week.
 - Doorloop hierbij telkens de procedure om de testsuite op te bouwen.
 - Maak nadat de testsuite compleet is de implementatie.

Stap 1: Opzetten van een project

NetBeans:

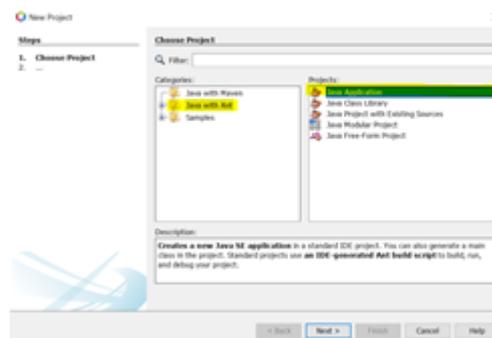
Maak een nieuw 'Java with Ant' project aan.

Visual Studio Code en IntelliJ:

Zie documentatie.

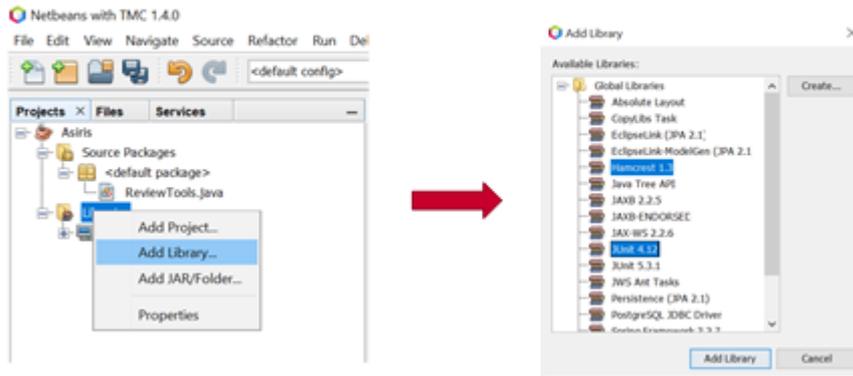
Tip:

Probeer eerst testen werkend te krijgen in een 'proof-of-concept' project voordat je testen gaat implementeren in een bestaand project.



Sommige IDE's werken iets beter samen met bepaalde versies testframeworks. Hierdoor is het handig met een 'leeg' project te proberen wat het beste werkt voor jouw IDE. TMCBeans die wij gebruiken, is een aangepaste Netbeans variant die niet fijn samenwerkt met JUnit 5. Je mag best voor een Maven project kiezen, maar dan moet je je testframework configureren in de Maven configuratie. Ant is het meest laagdrempelig.

Stap 1: Opzetten van een project



Programmeren 2

Let op: dit voorbeeld is gebaseerd op TMCBeans

21 oktober 2021

| 27

APPENDIX C: THE 'MILESTONE 6' ASSIGNMENT INSTRUCTIONS

Milestone 6 – Testopdracht.

Algemeen:

Lees voordat je aan de opdracht begint (niet op dezelfde dag dat je met je partner afsprekt er aan te gaan werken) als tweetal eerst de opdracht aandachtig door. Zijn er nog onduidelijkheden of heb je andere vragen over de opdracht, stuur dan een bericht via Teams aan Arno Broeders (geldt voor alle groepen). Ook in de vakantie is hij bereikbaar, reactie binnen één dag, als je geluk hebt snel, heb je geen reactie binnen een uur, wacht dan gewoon rustig af.

Voor de groepsopdracht moeten een aantal methoden gemaakt worden die verantwoordelijk zijn voor validatie en formattering van gegevens. Uiteraard willen wij als software-engineers een goede kwaliteit applicatie (Codecademy app) opleveren en hierin zo goed mogelijk borgen dat de validatie en formattering correct werkt en blijft werken.

Om deze milestone uit te werken zijn er 4 methoden in losse Java class-files aangeleverd, inclusief uitgebreide JML specificaties. Voor deze methoden moeten JUnit testen geschreven gaan worden.

Zoals we al weten uit de les, is testen een essentieel onderdeel van het ontwikkelen van een applicatie. Als je niet test, kun je niets zeggen over de kwaliteit van je product en heeft je product een grote kans om te falen (met vaak hoge kosten). Het wel goed testen levert dus een beloning op.

Door het afronden van deze milestone kunnen jullie als beloning een individueel bonuspunt verdienen op het programmeercijfer voor de Codecademy opdracht. Je verdient dit bonuscijfer per deelnemend koppel en niet meteen per groep! Als beide koppels in de groep hun opdracht inleveren krijgt iedereen in de groep dus het bonuspunt. Na het inleveren van deze milestone kun je de implementatie en de tests meteen één-op-één integreren in je groepsopdracht. Doe dit echter pas na het inleveren van deze milestone!

Hoe:

Dit is een 'Think-aloud pair-programming' opdracht zoals je ook bij de 'e12-weerstand' weektaak in programmeren 1 uitgevoerd hebt.

Je moet aan de volgende belangrijke voorwaarden voldoen:

- Een pair of koppel zijn **twee** studenten, dus geen drie of vier. Als je geen koppel kunt vormen neem je contact op met je docent. Uitgaande van projectgroepen van vier studenten, heeft iedere groep dus twee koppels.
- Je verdeelt de vier methoden waar tests voor moeten worden gemaakt over de twee koppels. Met twee koppels in jouw groep maak je dus complete testsets voor alle vier de methoden. *Hiermee is niet gezegd dat er maar vier methoden in het project getest moeten worden, kijk daar aandachtig waar je kwaliteit door middel van tests kunt borgen door zelf de specificaties uit de beschrijving te halen. Dit valt echter buiten deze Milestone opdracht!*

- Ieder koppel begint met het zo compleet mogelijk maken van **twee JUnit testsets op basis van de specificaties**. Je gebruikt hiervoor het **stappenplan** wat we ook in de les gebruikt hebben. Je vindt het op blackboard bij les 8. *Hoe pak je dit aan: Een praktische oplossing tijdens het maken van de opdracht zou kunnen zijn dat één student het stappenplan bijhoudt en de andere student daarbij aanstuurt. Na het afronden van één testset draai je de rollen om. Uiteraard mag je hier ook van afwijken.*
- De methoden met specificaties zijn aangeleverd in vier java bestanden. Voor deze milestone-opdracht verander je de klassenamen **niet** en laat deze methoden dus in de opgegeven klassen staan! Je download ze vanaf blackboard, ze staan bij deze milestone-opdracht.
- Voor iedere gekozen (in totaal twee) implementatieklasse maak je één testklasse met daarin alle tests voor die ene methode. Kijk voor de naamgeving van de testklasse in stap 3 van het stappenplan.
- Je maakt een schermopname van je ontwikkelomgeving tijdens het uitwerken en je neemt daarbij ook het gesprek met je medestudent op. Denk hardop en laat vooral blijken hoe je de verschillende stappen toepast. Laat vooral je onderbouwing horen waarom je iets doet. Val elkaar gerust in de reden als je denkt dat iets anders of beter kan.
- Pas **nadat** je de twee testsets hebt gecodeerd begin je aan de **implementatie** van de methoden. Je hebt dus eerst twee sets van falende testen op dat moment.
- De twee andere methoden (de methoden waarvoor je niet de testsets hebt gemaakt), **die implementeer je ook**. Je mag voor deze twee methoden zelf beslissen of je wèl of géén tests vooraf maakt.
- Het opzetten van het project en importeren van de libraries en de gegeven klassen hoef je **niet** te verwerken in de video-opname. Dit kun je van tevoren klaarzetten.

Inleveren:

- 2 testklassen met voldoende tests gemaakt op basis van de JML specificaties en het stappenplan.
- 4 implementaties van alle gegeven methoden. De 2 extra methoden mag je op basis van eigen inzicht coderen. Heb je hiervoor ook tests gemaakt, dan lever je deze ook extra in.
- Een link naar de video-opname van je ontwikkelscherm, inclusief audio waarop jullie samenwerking als koppel goed te horen is.

De inleverlink staat op blackboard bij de overige milestones. Van ieder koppel levert één student de materialen in:

- Laat in de omschrijving achter met wie je hebt samengewerkt.
- Laat in de omschrijving de url achter waar je video-opname te bekijken is, zorg dat deze een half jaar beschikbaar blijft en dat Arno, Ruud en Dion hier toegang tot hebben.
- Voeg een zip bestand toe met de uitgewerkte Java bestanden (tests en implementatie).