MASTER'S THESIS

TEACHING SPECIFICATION WRITING TO IMPROVE STUDENTS' CODE

Hermans, R.

Award date:
2021

TEACHING SPECIFICATION WRITING TO IMPROVE STUDENTS' CODE

by

Ruud Hermans

to be defended publicly on Wednesday June 9th, 2021 at 10:00 AM.

Open Universiteit
de beste! www.ou.nl

TEACHING SPECIFICATION WRITING TO IMPROVE STUDENTS' CODE

by

Ruud Hermans

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

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

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

ACKNOWLEDGEMENTS

In writing this thesis I have received different kinds of support, all equally valuable.

First, I would like to thank my thesis committee, dr. ir Harrie Passier, dr. ir. Sylvia Stuurman and prof. dr. Lex Bijlsma, for giving me the chance to graduate under their guidance, even though they were at capacity already. I also thank them and the rest of the QPED group for their guidance, review, and words of encouragement when I felt unmotivated.

Second, I want to thank my colleagues at Avans University of Applied Sciences, for obliging me in my research, even though it may have taken them more time than usual to perform our lessons, and for addressing the insecurities I had about my research. I explicitly want to thank my co-instructors in programming, for taking the time to think with me on how my research could best be performed in an already strange academic year.

Last, but never least, I want to thank my wife Myra, for suffering my anxiety about graduating, and supporting me in my endeavours to get my master's degree, while never judging me when I felt discouraged or unambitious. I could not have done it without you.

Contents

SUMMARY

The research described aims to improve students' understanding of the quality of their code through better analysis. Our approach was to teach them how to purposefully write specifications for public methods and test how this affected the quality of their code.

First, we created material based on four-component instructional design to instruct first-year Software Engineering students on how to write specifications, including procedural guidance. We asked students to practice with this method on supplied assignments, before giving them a large assignment they worked on together through pair-programming. Our observation of the pairs shows very few groups used our method, which was later confirmed in both evaluation and interviews.

To test whether the mere instruction without practice had influenced the students' quality of code, we reviewed the output of the large assignment through a standard model, focusing mostly on the external code quality. We did find a large difference in this area, but were unable to link this to the instruction, because the size of our test group gave us very little confidence in the accuracy of the data. The exam results, an indicator for general programming ability, did not show a big difference between the test and control groups.

On whether or not the method influences students' code quality in a positive way, the answer remains uncertain. The research does however add to the body of knowledge of software engineering by providing material for training on specification writing, with the accompanying procedural guidance being useful for both educational and professional use. We are confident that, if deployed correctly, the writing of specifications will positively impact code quality.

To answer our main research question, repetition is needed with some adjustments. A larger sample size would increase the confidence in our data, students need to be coerced into using the method, and in the setup it needs to be better considered how unintended factors may influence student performance.

Our method of teaching students specification writing can also be used when investigating whether better tests are actually written by students that have training in specification writing.

1

INTRODUCTION

As CS educators, we are both blessed and burdened by our experience in software engineering. Blessed because we have knowledge that we might transfer to students, but burdened by this same knowledge, as it prevents us from transferring it effectively, a phenomenon known as the *expert blind spot* [Nathan et al., 2001]. Experienced software engineers are used to working with a top-down approach, gathering requirements from users that are input for a high-level architecture, which in turn is used as input for the design of specific components, and so on. How we do this specifically is often not written up, as the process in time becomes so natural to the software engineer, that they have trouble putting it in writing [Buck and Stucki, 2000]. The novice, however, needs to understand this process and practice it assiduously to achieve the same level.

One skill so basic to experienced software engineers that it is often overlooked in teaching programming is *problem solving*: the ability to take a problem description in natural language, comprehend it, form a solution strategy, and transform it into a piece of code that will solve the exact problem described. Inexperienced software engineers have trouble with the comprehension of problem descriptions, but taking this step is in many cases not taught at all. A realistic problem statement often introduces a problem domain that is new to the programmer, even to experienced ones. After all, most applications are written to support business processes other than software engineering itself. A misunderstanding of the problem domain leads to bad solutions [Winslow, 1996]. For students, misunderstanding is an even larger factor, as they often read the problem description superficially, or have trouble relating the problem to one they know. With practice, understanding problem statements and specifying a solution for them becomes easier, but we believe teaching the writing of specifications to novice programmers directly improves their problem analysis skills, and as a consequence, improves the quality of their code.

This is not just conjecture. It was previously shown that novice programmers have a different idea of when their code is correct. Given a problem description, they consider the solution complete when it works for some examples, or in some cases, when it compiles [Kolikant, 2005][Edwards, 2004]. Entry-level programming courses increasingly teach students how to test their code, but there is evidence that even when testing is taught, students tend to only write happy path tests [Edwards and Shams, 2014]. The first step to remedy this is to teach the writing of specifications [Leventhal et al., 1994]. Previous work of the QPED research group, for which this proposal is written, offered the same conclusion [Bijlsma et al., 2021]. We want to contribute to students' ability to test by teaching them how

to write specifications, so that they may have a better understanding of the quality of their code through better analysis. It seems intuitive that the act of purposefully defining a specification will force the student to not only think about what they have to do to get it right, but also about how they could get it wrong, and prevent that.

The research described here shows how students were taught to write specifications and how this affected their programming skills. We provided novice programmers with procedural guidance that guides them step-by-step in analyzing a given problem statement written in natural language, and in writing the specification for the solution of this problem. We provided both detailed instruction on how to analyze a given problem statement, and how to write specifications in a specification language. We performed the research in the propaedeutic year of the Software Engineering program of the Avans University of Applied Sciences in Breda, The Netherlands. Given the institution's focus on the education of graduates who are employed as software engineering practitioners, and the lack of prior knowledge of formal logic, the specification language used is informal. The students selected for our research in general had no prior knowledge of programming before starting our program. At the point in the course they were taught our procedural guidance, they had merely learned the use of variables, command line input and output, control flow, methods, and debugging, all in the Java programming language. Our results are as such useful in the most basic of programming courses.

In the next section, we explore the research that has already been done in this area. Section 3 describes the research method used. In section 4, we present our results. In section 5, we discuss these results and in section 6, we present our conclusions and recommendations.

2

Related Work

Our work does not stand on its own. It builds on the large body of knowledge of programming pedagogy. In this chapter, we explore work that is related to ours and evaluate its suitability for students in the propaedeutic year of a University of Applied Sciences.

2.1. Acquisition of complex cognitive skills

Programming is a skill that synthesizes, among others, the knowledge of a programming language, the skill of writing code, quality-mindedness, and the concepts of abstraction and generalization to create solutions for often ill-defined problems. Learning such a complex skill is not merely the result of learning all its aspects separately. It takes a holistic view on learning to integrate these skills. Van Merriënboer and Kirschner posit such a holistic approach, called *Ten Steps to Complex Learning*, which is a modified, practical version of the *four-component instructional design model* [Kirschner and Van Merriënboer, 2008]. The model describes how the acquisition of complex cognitive skills is best organized in a program. Complex cognitive skills are those that require the "integration of knowledge, skills and attitudes; coordinating qualitatively different constituent skills; and often transferring what was learned in school or training to daily life and work." The focus is on learning tasks that integrate all these aspects. The model builds on the proposition that a blueprint for the learning of complex cognitive skills can be described by four components: learning tasks, supportive information, procedural information, and part-task practice.

Learning tasks are those activities a learner undertakes that require them to integrate skills, knowledge, and attitudes in a real-life task. *Supportive information* supports the learner in those tasks for the aspects that are not routine. *Procedural information* supports the learner with steps to take for routine aspects that are always performed in the same way. *Part-task practice* repeats routine aspects of the complex cognitive skill. While programming is indubitably a complex cognitive skill, the challenge is to map the specific sub-skills of programming to either routine or non-routine [Van Merriënboer and Kirschner, 2017].

2.2. Novice programmers need problem solving strategies

A novice programmer is someone who has just started their first learning experience in programming. Our aim as CS educators is to start the novice programmer on his path to

becoming an expert programmer. According to Winslow, this takes approximately ten years [Winslow, 1996]. Since the institution only has the student at their disposal for four years, the curriculum should focus on providing students with the skills to guide their growth after graduating. Part of this can be the development of methodical work habits, in which procedural guidance plays an important role. At some point, students will be experienced enough to not need the step-wise instruction, but will hopefully have grown accustomed to thinking ahead when starting on a new problem.

Novices share characteristics independent of their field of expertise. General psychological and pedagogic studies are in that sense helpful to understand the needs of novice programmers as well. Winslow states most research differentiates between problems and tasks. A task is a goal with a known solution, and a problem is a goal without a familiar solution. Novices tend to encounter more of the latter and thus use more general problem solving strategies. Experts, in general, recognize patterns in problem statements and apply more specific strategies to solve recurring problems. These strategies are known as plans, templates, schemata, or idioms [Clancy and Linn, 1999]. If taught to the novice learners they can be helpful, but in our exploration of related work we did not encounter any for the analysis of extensive problem statements. This may be because generalizing a problem statement to a more generic variant that is needed to apply a template, already requires an analytic skill that novices do not yet possess. We can however make the process experts use to approach a new problem statement explicit, so that the novice can benefit from the methodological approach as well.

Students who can translate problem statements to code still have misconceptions about the quality of their code. Kolikant wrote about how standards for correctness differ for novices as compared to professionals [Kolikant, 2005]. For example, students felt a program was correct with as little evidence as that it gives some output for any input, even if it is not the expected input, or that they are content with it just compiling. The findings are corroborated by Edwards, with him concluding that students should move from *trial and error* to *reflection in action* [Edwards, 2004]. Reflection in action is using analytical thinking to solve a problem when one occurs, instead of just changing around some code to see if that solves the issue. Edwards has his students write tests to promote reflection in action. He not only assesses a student's code, but also the completeness with which the student has tested it. This requires the student to think about much more than just the happy path, and improves the quality of their code as a result. The same result might be reached when students are compelled to write specifications, using procedural guidance which sets them to analytical thinking a priori.

2.3. Procedural guidance in problem solving

Thompson takes Polya's seminal work *How To Solve It* [Polya, 1957] and applies it to programming in his method *How To Program It*, as an answer to the question he hears from a lot of beginning programmers: *"Where do I begin?"* [Thompson, 1997]. He provides a four-step process for solving problems, which are *understanding the problem, designing the program, writing the program, and reflection.* The first step, *understanding the problem*, leads to the definition of one or more specifications, depending on the size of the problem and the nature of the solution. When taking this step, the programmer clarifies the original problem statement, to ensure the goal is clear. Following this, the programmer specifies the signature of their solution: they choose an appropriate name, and defines the

inputs and outputs. In the definition of the corresponding specification, the programmer thinks about possible special conditions for the inputs and outputs, which already forces him to consider non-happy paths. Thompson's method continues with a step in which the method body is designed, which is not relevant to our research. The specification step is of a very high abstraction level, describing the aspects a programmer should consider, but giving little guidance on how specification should be done. In his article, Thompson does not give an example of this either.

A similar approach is the *design recipe* proposed by Felleisen et al.[Felleisen et al., 2018]. It is step-by-step instruction introduced with, but not limited to, the Scheme language. Similar to Thompson's process, the design recipe starts with understanding the problem and defining the function. In addition, Felleisen emphasizes examples: finding the ones available in the problem statement, validating outputs for certain inputs, and finding new examples. In this process the programmer can already stray off happy-path implementations, and consider the expected output without the temptation of trusting the code's output. Finally, giving examples serve as specification, making abstract and ambiguous specification concrete. As with Thompson's work, Felleisen's design recipe is defined on a high level. The first phase, *Contract Purpose and Header*, has semi-concrete procedural guidance:

1. Choose a *name* that fits the problem;
2. Study the problem for clues on how many unknown 'givens' the function consumes;
3. Pick one variable per input; If possible, use names that are mentioned for the 'givens' in the problem statement;
4. Describe what the function should produce using the chosen variables names;
5. Formulate the contract and header (in this book specifically, in a Scheme definition);

The last step of this phase is the only language-specific step, but the definition of a contract is similar to Meyer's *contract* (precondition and postcondition) [Meyer, 1992]. The difference is that in Felleisen's recipe the contract is informally specified.

The second phase, *Examples*, has similar guidance, but most importantly does not give points on how to 'make up examples'. On this, the recipe needs clarification.

2.4. Informal specifications

Buck and Stucki base their coursework on the notion that students have to start small and work towards larger and more integrated assignments. They believe the correct way to teach programming is by starting off with small programs, even just one-liners, without having to worry about the application design. Their believe is that the novice is not yet at the cognitive level needed to properly decompose a program in procedures and writing correct specifications, and as such should not be burdened with this when just starting to code. Buck and Stucki mention beginning students should not be involved in writing specifications, but rather learn to read them first, so that the specification may function as assignment descriptions. While they use formal specifications, they do provide informal specifications, which are meant as support in learning to read and understand the formal specifications. They mention the informal specifications contain ambiguity, but convey

intuition to the student about the formal specification. In contrast, we believe beginning students will benefit from writing informal specifications, as this process will enhance understanding of the task at hand, and their ambiguous nature allows for beginner's mistakes [Buck and Stucki, 2000].

To deal with the ambiguity of informal specifications, tests may be provided as additional specification. Specification-Driven Development combines up-front specification with test-first programming. It states that incomplete specifications should be supplemented with tests that provide examples. In this way, tests are considered specifications themselves [Ostroff et al., 2004]. However, we believe this will not be helpful to novice programmers, as they tend to only write happy path tests [Edwards and Shams, 2014]. Our work does not include the teaching of testing.

3

METHOD

Even though the body of knowledge on problem solving is large, we have not found previous work that describes how engineers should transform a problem statement that is written in ambiguous natural language to a method specification in an informal language. As shown, this should be useful for novice Software Engineering practitioners. Our research focused on describing these steps, and on novices in an educational environment specifically. The research question answered in our research is:

In what ways do students benefit from education in writing informal specifications?

This question was divided in several sub-questions:

- What does effective education in specification writing look like?
- What difference can be seen in students' approach to problem-solving with and without this education?
- How is the quality of students' work affected by this education as compared to the work of students without this education?
- What is the difference in exam scores between students with and without this education?

For each sub-question, we explain the rationale of the question, and the method of research we used to come to an answer.

3.1. WHAT DOES EFFECTIVE EDUCATION IN SPECIFICATION WRITING LOOK LIKE?

To answer this question, first we define effective education as education that, using the limited time at hand, trains the students in such a way that they can achieve the intended goals of the program or course. For this, additional content was developed for the first programming course in the first-semester of the propaedeutic year of the Software Engineering program of the Avans University of Applied Sciences in Breda, The Netherlands. The following sections discuss the design of the material used to teach students specification writing. It shows how the work of van Merriënboer influenced the educational design.

3.1.1. Design principles

Writing good specifications is a tricky endeavour even for advanced engineers and as such can be considered a complex task. Currently, the program is set up objective-based, in that students are expected to reach set objectives, which together indicate a level of comprehension. Task-based education, as compared to objective-based education, is better suited to teach complex tasks such as specification writing, as these need the integration of knowledge, skills and attitudes and are not just learned by stacking compartmentalized pieces of knowledge [Van Merriënboer, 1997]. Four-component instructional design (4C/ID) is one model that describes how educational programs are designed task-based.

4C/ID describes four components: learning tasks, part-task practice, supportive information and procedural information. Learning tasks are those that integrate skills, knowledge and attitudes in realistic tasks. In our context, for example, this can be a project in which students work together in a Scrum team to create a software application. This is regarded as a complex learning task and suited for a later part of the curriculum. The creation of a piece of software to satisfy a simple problem description is a learning task better suited for the first semester of our program. This is practiced thoroughly by students in their first half year.

Learning tasks are the base to which the other three components are connected. It is supported by supportive information, which, if we consider our first half-year, is the classroom instruction provided, but also any information that is given in a different form, be it the Massive Open Online Course (MOOC) used, short clips provided or on-demand instruction when students are practicing. Supportive information is new information to the student, from which cognitive schemas are formed: abstractions of the knowledge and examples that can be applied to new situations. A good example is the use of the for-loop in programming. A student has formed a cognitive schema when they can apply the concept to a, for them, new context.

The automation of schemas, which ensures a student performs the part-task without giving it much thought, is stimulated by part-task practice and supported by procedural information. Practice of the part-task in our context is done by supplying plenty of exercises in the MOOC that focus on the part-task. Procedural information gives students guidance to first learn the task, and structure to fall back on when the student does not fully grasp the task yet. The entry-level programming course lacks procedural information, for example for the analysis of a problem statement and subsequent specification of a method that solves that problem.

To teach these students how to write specifications we created new supportive information in the form of an additional lecture, provided plenty of part-task practice opportunity through the exercises in the MOOC used in the course, and designed procedural information for the step-wise analysis of a problem statement and subsequent written specification. Although our work focused on teaching specification writing in a University of Applied Sciences, it was a joint effort with the *Procedural Guidance* research group of the Open Universiteit of The Netherlands.

3.1.2. Learning tasks

The Software Engineering program is designed around competences picked from the European e-Competence Framework. This framework describes the competences an IT-professional can have, irrespective of the context they are used in. The competences are in that sense

representative of the workplace tasks and suitable as input for learning tasks. The competence relevant to this research is Application Development. This is a multi-part competence, with skills ranging from programming, to testing, but also specification, design (in part) and analysis thereof.

It is uncommon for instruction in the program to explicitly connect the learning task to the workplace environment, but the developed instruction does this, as recommended by [Frerejean et al., 2019]. We expected some students would not understand why it is necessary to analyze and specify for a complex assignment. To prevent these students from getting the idea that this is just an educational exercise, it was deemed necessary that the material shows a realistic scenario in which analysis and specification takes place. This is included in the supportive information, as the program had no place dedicated to showing the bigger picture of software engineering (yet).

3.1.3. PART-TASK PRACTICE

In the programming course, students are focused on part-task practice for coding. The subject of design is not touched on in the course. Its aim is to give students enough proficiency in the Java programming language, so that they can create very basic object-oriented programs with console input and output. This requires a lot of repetition, which is achieved by having them solve more than two hundred programming problems.

For our research, we selected suitable exercises from this large set of programming problems, on which students could practice analysis and specification. These exercises start simple, with a given method signature, but progress to more complex programs for which multiple methods are needed. Appendix A lists the suitable exercises for part-task practice. The solution correctness for these exercises is automatically unit tested and awarded points to, but instructors need to review the specifications manually, to provide feedback to students and keep them practicing. Appendix B and C show two of the larger weekly assignments that are also suitable for part-task practice.

3.1.4. SUPPORTIVE INFORMATION

Supportive information is provided in the form of a slide deck for instruction, a recording of this instruction, so that it can be used at a later point, explanation and substantiation that accompanies the procedural guidance, and references to the JavaDoc technical resource at https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html. All this information was communicated through Blackboard, the Learning Management System (LMS) the institution uses. It is also provided as appendix to this thesis.

3.1.5. PROCEDURAL GUIDANCE

The procedural guidance conveys how an experienced software engineer approaches specification of a method. To create it, we analysed two previously unknown assignments and consciously wrote down the mental steps taken in the analysis, and how parts of that analysis link to the specification. This formed the input for what needed to be considered for the procedural guidance. The goal of the analysis is to come to a specification, so that students think about their code's intent up front. With that in mind, and keeping in mind that these students are studying to become Software Engineering practitioners, we decided for the specification to conform to the Javadoc standard as closely as possible. This ensures students learn to document code in the industry standard, and benefit from tools and re-

sources already widely available. As a result, specifications can be made directly available in the program code.

In consciously analysing the assignments, however, it became clear that pre- and post-conditions are the aspects of a specification most beneficial to thinking about code output for different inputs. If students are to benefit from specification, it needs to trigger them in considering these aspects. Javadoc has no explicit tags to specify pre- and postconditions, so for this purpose the *@requires* and *@ensures* tags from the Java Modelling Language (JML) were added. In general, more than one set of pre- and postconditions exist, for example when defining non-happy paths. Groups of these are separated with the *also* tag, prepended with @, so that code formatting keeps it in place. Although we adopt tags from the JML, we do not use it to its full extent. JML allows for automatic code verification through formal specification, but this is not taught in our work, as its use is of an academic nature and not commonly used by practitioners. For this same reason, we do not consider the *@assignable*, *@pure* and *@invariant* tags. If two tags exist in Javadoc and JML that have equal meaning, the Javadoc version is preferred. The language used in the specification is of an informal nature.

Private methods were not considered in our work. Our focus was on external specifications, visible to clients of public methods. We did not consider internal specifications, neither for private, nor for public methods. External specifications can only refer to information that is visible to clients, not implementation details. The students who are part of this research only learn the basics of object-oriented programming in the first half year. Basic design is only introduced in the second semester of the first year. Part of design is learning how to make the choice between making a method public or private. Both modifiers are taught in Programming 1, but making a proper choice between them is not. As such, we only considered external specification for public methods.

The procedural guidance that helps students write informal specifications for public methods is available in Appendix D. Its intended purpose is to trigger students to think about all the preconditions their code needs to handle, so that they may code defensively. Although the translation of a specification to code is not part of our work, we hypothesize the procedural guidance will already impact the quality of their code in a significant way.

3.1.6. Validation

Two teachers of the program, as well as five members of the QPED research group, reviewed the instructional material before its use. The instruction was validated by performing a survey with all participants on the conclusion of the course. The survey design is found in Appendix E. After the survey results were in, eight students were interviewed in a semi-structured interview to deepen our understanding of the feedback on the effectiveness of the additions to our programming education.

3.2. What difference can be seen in students' approach to problem-solving with and without this education?

Our expectation going into this research was that students who have received education in specification writing approach a programming problem differently than their peers who

have not received the same instruction. To verify this, we employed the *observation of behaviour* method from *The Think Aloud Method* by Van Someren et al. [1994]. In this method, the student is required to verbalize their thoughts by thinking out loud while working on a problem. This is an unstructured technique, in that it does not steer the observed student in any way other than giving them the initial assignment description. Usually an *experimenter* prompts the student to think aloud when they fall silent.

To test our hypothesis, the additional training in specification writing was provided to a test group of about a third of the first-year population. This test group was not selected, but consisted simply of one of the classes that students were distributed to at the start of the academic year. The control group did not receive the same education. We then provided an exercise to both the test and control groups in which they record themselves pair programming on a problem that was specifically selected for this study, and is made available in Appendix C. Participation in this assignment was not mandatory, but stimulated with a bonus point for the exam, as was the case for all weekly assignments.

Employing the pair programming method made it possible to work through the exercise without an experimenter present, the idea being that working together would prompt the students to think aloud. We had students record themselves while working together on the problem, so that the recording could be analyzed at a later point.

We expected two aspects to take form in the thoughts the students exchange on the analysis of an assignment. First, the use of the procedural guidance provided. Second, a methodical line of thinking, in which they are obliged to by following the guidance. As Edwards stated, and in our experience, novices generally do not exhibit a methodical approach to programming when they are not taught to do so, so we expected this to be a notable difference [Edwards, 2004].

The recordings were analyzed for the entire test group and part of the control group, to detect if a difference in behavior on the whole could be seen. To validate if differences found were based on the additional education rather than other factors, we used unstructured interviews with eight students of the test group to assess the underlying motives for shown behavior.

3.3. How is the quality of students' work affected by this education?

Another hypothesis we had is that students who are educated in specification writing produce code of better quality than their peers who have not had the same education. Better quality in this case means more likely to solve the problem they were asked to solve, and more robust against bad input. We conjectured this to be an effect of the act of specification, in which a student is set to think about what their code should do before actually writing it.

We used code reviews using a standardized model to assess the quality of the code students wrote for the assignment available in Appendix C. The standardized model is available in Appendix F. Because the specification focuses on public methods, we primarily looked at the difference in quality of the public API, but the assessment model also scores on internal code quality, being the implementation of the method. To validate deviations between the two groups are not caused by other variables, such as the amount of practice, prior knowledge, or instructor they had, we plotted the resulting marks against each of

these variables, as well as against if they got the instruction on specification writing or not.

3.4. What is the difference in exam scores between students with and without this education?

Similarly to the previous research question, this research question is to test the hypothesis that students who have been educated in specification writing perform better on exams. Each iteration of the course is wrapped up with an exam, and this iteration was no different. As such, no additional preparation work was needed to answer this research question. Again, to validate deviations between the two groups are not caused by other variables, we plotted the resulting marks against each of the variables as described in the previous research question.

3.5. Research method and ethics

We based our research methods mostly on a division of students in a control and a test group and the resulting difference in their behavior, marks, and quality of code. In addition, we wanted this to be a partial blind study, and as such not let the students know that they received extra education. Our goal was to prevent the positive bias that comes from self-selection, and as such we have not asked students to volunteer for our study. We have to consider if our choice of methods is ethical. The students who are instructed in specification writing might have an edge on the students who have not been instructed similarly. On the other hand, as this was additional education on an existing course, they may have been disadvantaged by it.

We justify our approach in two ways. One, we did not know up front if the additional education increases a student's performance or degrades it, and two, the control group was taught in the same way as the previous iteration of this course, which we already considered quality education. To prevent the effect that observed subjects behave more positively, commonly referred to as the *Hawthorne effect* [Franke and Kaul, 1978], we did not discuss our research with students at all. The control group has received the same instruction on completion of the study. If the results of our research show that procedural guidance significantly improved the results of our students, the control group will be allowed to retake the exam.

4

RESULTS

In this chapter we showcase the results of our study, which we will discuss in the next chapter.

4.1. EDUCATION OF SPECIFICATION WRITING

In this section, we try to answer the question: What does effective education in specification writing look like? A group of 35 students received the one and a half hours of instruction on analysis and specification of methods on the 30th of September 2020. Due to COVID-19, this instruction was given online through Microsoft Teams. In the instruction we introduced the procedural guidance, the role of specifications in software engineering, and the exercises students could practice on. During the training, students were asked to practice with the procedural guidance on an assignment specifically designed for that moment. The instruction was given as an addition to the existing Programming 1 course. The control group of 84 students followed the same course simultaneously, but did not receive the additional instruction or material.

4.1.1. SURVEY RESULTS

The survey in appendix E was added to the general survey on the whole course and sent to 119 students. The section specific to specification writing was only available to the test group. The total of survey responses is 81, of which 28 responses from the test group and 53 responses from the control group.

ON SPECIFICATION WRITING

The following questions on specification writing were part of the survey. Participants scored their level of agreement with the statement on a scale from 1 to 5 including.

I found *analysis and specification* an interesting part of the course. Average score of 3.3.

The quality of the materials provided on Blackboard for *analysis and specification* was high. Average score of 3.8.

The effectiveness of the additional training on *analysis and specification* was high. Average score of 3.3.

During the training on *analysis and specification* it was clear how the subject relates to working as a programmer in a company. Average score of 3.6.

It should be noted that as compared to the average score of 4.6 for the course as a whole, these numbers are low for the test group. The quality of the materials provided are not a matter of concern, but students not considering analysis and specification to very interesting, to the point where they barely gave it a passing mark, is. At the same time, they did not think the effectiveness of the additional training was high. Just 9 out of 19 respondents in the test group indicate they used our materials during exercises. Of these, the interviews show that at least some students mistake a step-wise approach of their own device for our procedural guidance. Clearly, work on the effectiveness of our training is needed when using it in other contexts. It should be noted that students' concept of 'effectiveness' can be a different one than our interpretation.

The answers to the survey's open questions do not set us in the right direction for changes so that the material we use might be more interesting. They do however hint that the effectiveness of the education on specification writing as a whole might be increased with more practice opportunity. When asked what could be changed for the better, students said:

"Maybe some more interaction, and practice through exercises."
"Explanation through exercises would make it a lot clearer."
"A mandatory exercise that counts for bonus points on the exam to stimulate practice."
"There's no specific exercise to apply [the new knowledge] on."

It seems students prefer a dedicated exercise (and feedback on this exercise) to integration with the other programming assignments. Exactly such an exercise was provided at the end of the instruction, but at the time, it was obvious a lot of students chose to continue working on the MOOC programming assignments, instead of practicing specification writing.

Although just one student of the test group was not present when the extra instruction was given, multiple respondents indicate they have no knowledge of ever receiving it. This also attests to a certain lack of effectiveness of the training. This no doubt was amplified by the lecture taking place online. The timing of the lecture will also have influenced effectiveness of the instruction: it was provided additionally to existing material, on the same day an introduction in object-oriented programming was given. Of the positive remarks on the training, what most stands out in multiple answers, is that the clarity of explanation of the training was high.

Other notable differences in the evaluation between the test and control groups

The 53 participants of the control group only answered the general survey questions and did not see any questions on specification writing. We can, however, note some interesting differences between the two groups. For one, two statements on support during practice were scored from 1 to 5: "I was sufficiently supported while practicing programming." and "I know where to get support while practicing programming.". For the test group, the scores on these answers were 0.3 points higher (4.6 for both) than for the control group (4.3 for both). A suitable explanation might be that the lecturer for the test group was not the same as the lecturers for the control group, but the difference could also be caused by the extra

instruction the test group had, and the availability of procedural guidance to fall back on.

Another notable difference is how students rank what skills were hardest to learn. The skills to rank were:

1. Understanding an assignment (What is asked of me?)
2. Convert an assignment to an algorithm (How would I do it on paper?)
3. Convert an algorithm to code
4. Debugging a problem in code

While for both groups the first and third ranked skills were the same (skill 2 and 3) the test group on average ranked skill 1 as the second hardest skill to learn, while the control group ranked the same skill as least hard to learn between the four. It is possible our additional instruction has made the test group aware that understanding a problem statement is harder than it seems and that they are now 'consciously incompetent'. It is also entirely possible that they link this skill to the additional instruction and thought it was hard to follow, although students did not indicate this in the interviews.

4.1.2. INTERVIEWS

We interviewed 8 students from the test group, among whom was a survey participant that had indicated they did not remember receiving extra instruction on specification writing. Obviously, the first item in this specific interviews was figuring out if they really missed the instruction, even though they were registered as having been in attendance. This one student could not recall having attended the instruction, even when prompted. As such, this student had no further comments on the training, but in the interview, did implicitly mention justification for the existence of the material:

"[With the Programming 1 assignments] we just start and see what happens. Now I notice up-front planning is useful and I work more thoughtfully. It would be nice to get an assignment in Programming 1 that shows you can't complete it if you didn't plan upfront."

A common theme in the other interviews was the recognition that writing a specification by following our procedural guidance is useful for understanding an assignment. However, as the survey results show, it was not employed much. On why this was the case for them, one students had said:

"It is logical [specification writing] is taught, I get why it is useful, but it is also an extra step. I don't supply comments, because it costs extra time that I do not wish to spend. Using the procedural guidance slows me down."

This mentality was shown by three of the other interviewees as well. Writing comments and especially Javadoc was perceived as boring and superfluous, as, in their understanding, it contains the same information as the code. They thought the immediate reward that writing code has, is something that writing specifications lack. These students have not yet had the pleasure of having to maintain other programmers', or even their own code.

On the subject of how we can make the topic more useful to them, students had this to say:

"Make it shorter."
"Provide a cheat sheet for it."

In these interviews, again, it was noted that just providing the theory and not having student practice on an actual assignment did not stimulate the use of the procedural guidance. This does paint the picture the extra instruction was at least forgotten in part.

An interesting note from one of the interviewees was the idea that apart from just the textual problem descriptions in the MOOC, some students may also benefit from very small exercises, in which they would practice a program structure (like an if statement) over and over, without context and with just different values. They saw a parallel with math in this sense. It seems some part-task practice can be added to cater to students who have trouble grasping these core concepts.

4.1.3. Conclusion

What does effective education in specification writing look like? The material created to teach students specification writing consists of a slide deck for instruction, a selection of existing and new exercises to practice on, and procedural guidance. The material itself is considered of sufficient quality, but the retention of the instruction, and perhaps the retention of the knowledge itself, is lacking.

Students did not themselves choose to apply the knowledge, at least in part because they find commenting code to be superfluous. In their opinion, it is just restating what the code does. This, perhaps, can be explained with how novices comment code. Novices tend to comment code syntactically, in that they explain how the *programming language* works, while experienced software engineers tend to comment code semantically, in that they explain how the *program* works [Riecken et al., 1991]. The latter, obviously, makes comments a useful addition to code. An addition to make our education more effective, would be to adopt this purpose of commenting code in our material.

A second addition that would add to the effectiveness, is to stimulate students to practice by making at least some exercises either mandatory, or have them count towards the course grade. Students signalled this is necessary as incentive to put time in practicing this skill. It is our opinion that this, and the clarification on the purpose of comments, not just specification, would make the education sufficiently effective.

4.2. Differences in problem-solving

In this section, we answer the question: What difference can be seen in students' approach to problem-solving with and without this education? To this end, we collected over 85 hours of footage of pairs of students working on one large assignment together, of which 33 hours from the test group and 52 hours from the control group. We analyzed all of the test group footage and 18 hours of the control group footage, to discover if a difference can be seen in their approach to problem-solving.

The obvious question to answer is whether any of the pairs used our material in an obvious way, such as using the procedural guidance to analyze the assignment. None of the

pairs used our material from the start, but there was one pair that did take to the guidance when struggling with naming a method. Although this was not its intended purpose, the guidance did aid these students in producing a name. When asked why they didn't use the guidance from the very beginning, these students answered:

"We were eager to start coding, it was a big assignment. These weekly assignments take a lot of time."

It seems even the students who did use our procedural guidance, did not recognize upfront analysis can save time. Although, in several recordings it is seen that students do analyze the assignment upfront, but not in a way we expected. What we saw is that students made a direct step from problem statement to algorithm design in pseudocode. What is especially interesting about this is that we did not teach them to do this, but rather seems to be a strategy they developed over time. Whether that strategy developed while working on earlier programming assignments, or perhaps even before that, needs to be investigated further. Perhaps its origin can explain part of the survey results, in which it was shown that students from both groups found going from an assignment to algorithm the hardest part of programming. One student in particular mentioned in the interviews that, when we referred to our procedural guidance, they understood this to mean their own step-by-step approach to analyzing an assignment. If other students misunderstood the term in the same fashion, our results for use of our materials may be worse than initially thought.

Other than the one student pair that partially used our procedural guidance, we did not find significant differences in the problem-solving approach of the test and control groups. For example, in both groups we noticed the alternative standards for correctness students have, as described by Kolikant [2005], which are not affected by merely offering training in specification writing:

"Let's see if it works, put a couple of nice numbers in." After trying one example: *"I think it works!"*

Different group, after trying just one example: *"It works!"*

In both groups we observed most pairs:

- jumped straight to designing the algorithm for the solution.
- had different interpretations of the assignment between the two students.
- did not think twice about the naming of their methods and classes.
- did not think about what input would lead to an error state.
- did not realize we asked for a pure function.

The assignment explicitly calls for thorough analysis before coding, but most pairs can not resist the urge to start writing code after reading the assignment. This behavior is seen in both the test and control groups. When pairs did spend more time on their analysis, this did not necessarily lead to better code. In both groups, we saw a lot of cases of one student

reading the assignment description aloud to the other student. This seems to lead to superficial understanding of the assignment in both the reader and the listener. For example, in one case a pair of students explicitly read aloud the assignment called for a method, before immediately ignoring that and starting off on designing a class. This was seen for many pairs. It could be these students did not realize the assignment was not coupled to the topics taught that week. In class, they had just continued on object-oriented programming, which could have led them to assume this knowledge should be used on the assignment. For example, students mention:

"Use objects, otherwise you throw what we learned straight in the bin."

"They probably want us to use OO."

Similarly, for nearly all the groups that did not perform any analysis up front, we noticed that at some point in their discussion, they would speak about a term but have a different understanding of its meaning, a direct effect of not discussing the assignment up front to remove ambiguity. For example, in multiple groups the term 'nearest' lead to misunderstandings about whether this meant rounding down to the nearest possible value, or if rounding up should be considered as well. Because the students did not discuss up front what their definition of this term was, this led to extra time needed during the assignment. On a related note, these types of misunderstandings were not just seen for the interpretation of the assignment, but also for what the purpose of the part of code they were working on was. In one such instance, a long discussion took place about what we recognized to be two different purposes, until one student exasperatedly called out:

"I don't want to get the multiplier from this, I want to get the nearest value!"

We saw this type of discussion in both the test and control groups, which seems logical when one of the intended purposes of up front analysis is to clear up ambiguity of the problem description. In general, it does seem the pairs that discuss the algorithm design after reading the assignment have less misunderstandings when coding.

In both groups the students did not pay much attention to the naming of their methods and variables, but more than that, their attitudes seem to indicate quality of the assignment code did not matter much to them at all. This is perhaps to be expected; the code is after all not to be reused or expanded upon. However, for a lot of pairs, it seems quality is seen as somewhat of a time sink:

Student 1: "[Method name] needs to be clearer." Student 2: "Yeah, but a name is a name"

"I don't care what the [variable] name is"

Student 1: "I can read the code like this, so it's fine by me." Student 2: "I'm hungry anyway."

In multiple cases, the students even specifically mention they build something in to please the instructors:

Student 1: "Does it say Javadoc is mandatory?" Student 2: "No, but I think [the instructors] would rather see that."

"Instructor is going to be happy we're using a method for this!"

"In the training they were adamant we need to discuss up front, so let's do that."

Some remarks indicate students at least have an intuition the quality of their code needs improvement, but are impotent to do this themselves:

"We can probably do this more easily, but it works."

"The code we put most effort in, is the code we do not end up using."

"Next training we'll be the example of unclear code."

"I think we're doing this very unwieldy."

Lastly, the way students communicate about their code is very different from how experts discuss:

"We want to use another class, that's logical. What do we want in that class?"

"[The IDE] says this is an unnecessary import, because it is imported by default. Let's keep it in to be sure."

"How many classes will we work with?"

In the interviews we conducted, similar attitudes are shown. One student thinks his own code was 'spaghetti-code', but laughed if off, as if we could not expect more from him. Another pair of students explicitly mentioned that they only worked on the code so far that it covered the assignment criteria, because:

When the code covers the assignment criteria, it covers our criteria.

This seems to be further evidence students do not consider code quality to be more than the minimum properties code needs to have to cover assignment criteria.

4.2.1. Conclusion

In conclusion, what difference can be seen in students' approach to problem-solving with and without education in specification writing? Unfortunately, no real difference is shown, other than the occasional mention of one of the aspects of the training. The general attitude of students seems to be one of writing code to complete the assignment criteria, while building in quality is not on their (spoken) minds. While disappointing, this does not entirely come as a surprise. As these novices have just begun coding, they seem happy enough that they reach the goal of the assignment, regardless of how they did it.

In the next section, we will discover whether there was a difference in quality of the resulting code.

4.3. QUALITY OF CODE

In this section, we answer the question: How is the quality of students' work affected by this education? Although the previous section shows student behavior does not differ between the test and control groups, a difference might still be detected by reviewing the code they made. We code reviewed 38 attempts, of which 14 by the test group and 24 by the control group. In general, we noticed most attempts took an object-oriented approach, instead of supplying the method the assignment asked for. We assume students linked the assignment to the topics of that week, as was the case for earlier assignments. However, they did not apply that week's topic of specification writing to this assignment, so our assumption might be false.

We assessed both the external quality of the code, being that which can be publicly perceived, and the internal quality, being that which is concealed from the public view: the internal implementation of the code. When scoring on all criteria, we see an average score of 3.8 out of a possible 18 points for both the test and control groups. In that sense, no difference can be seen. However, when looking at just the criteria for external quality, which are expected to be positively influenced by our training, we see an average of 2.1 out of a possible 13 points for the control group and an average of 2.7 for the test group. It seems the test group scores 30% higher for external code quality. When breaking down our results for external quality (as shown in table 4.1, see appendix F for full criteria description), we see the test group scores higher on every criterion, except on, arguably the most important one, correctness.

Table 4.1: Found instances of external quality criteria between test and control groups in percentages.

Item	**Test**	**Control**
Method	29%	21%
Method name	14%	4%
Comments	7%	0%
Guards	7%	4%
Output	21%	0%
Input	21%	13%
Input name	14%	8%
Correct	29%	42%

Looking just at the groups that produced correct code, the average numbers are widely different. For all criteria, the test group scores 8.3 points, against 5.1 points for the control group, and for just the external quality criteria, the test group scores 6.8 points, against 3.9 points for the control group. Interestingly, the test group pairs that produced correct code did so in an average time of 117 minutes, while the control group average was 155 minutes. If the test group spent more time on analysis and as a result produced a solution faster, this difference might be related to our study.

All of the students in the test group that produced correct code followed the training on specification writing in full. The test group had the same instructor throughout the course. To determine if other factors than our instruction might be at play in this difference, we

plot the amount of practice and prior knowledge against the results of our code review. The amount of practice is measured in two ways. One, through the progress in the MOOC used in the course, by percentage, and two, by the amount of weekly assignments handed in. There are six weekly assignments in total, that more or less combine all topics of the MOOC for that week. In figure 4.1 we see the control group is further along in the MOOC, with an average completion rate of 75%, while the test group has an average completion rate of 65%. Even when taking the uncertainty of the data into account, the MOOC completion rate does not seem to explain the difference.

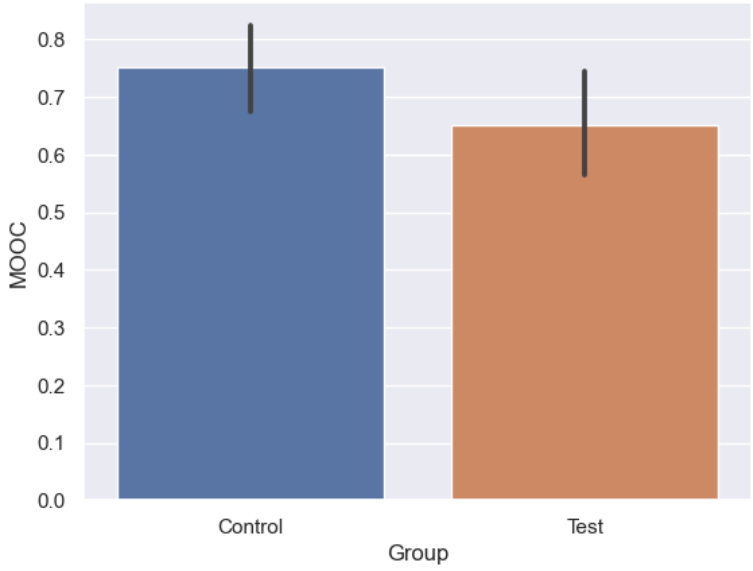


Figure 4.1: MOOC progress split by group for students with correct implementations

When looking at practice as measured by the amount of weekly assignments handed in, the same picture is painted. As shown in figure 4.2, the control group on average handed in more weekly assignments.

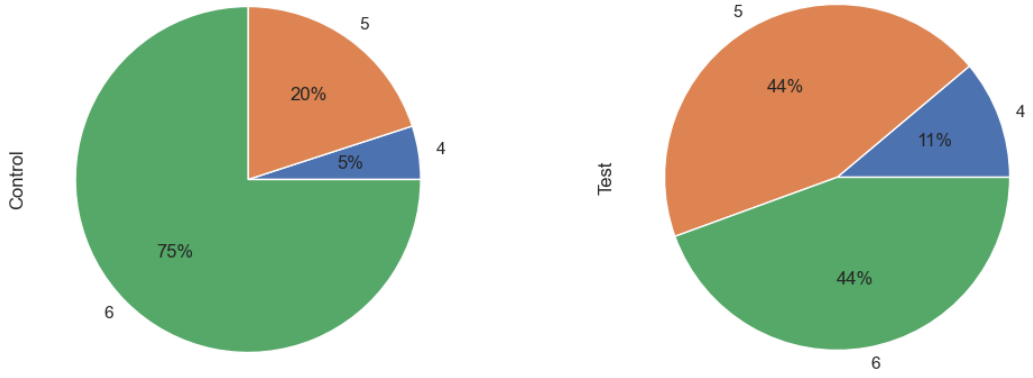


Figure 4.2: Number of weekly assignments completed split by group for students with correct implementations

It seems the amount of practice does not positively correlate with the score from code reviews. When looking at previous education, we do see a big difference between the

groups. In figure 4.3 we see that the test group consisted of students with havo as previous education, with the remainder coming from vwo. For the control group, we see that the group for 35% consists of students with previous programming experience, either from their previous education, or because they restarted the program.

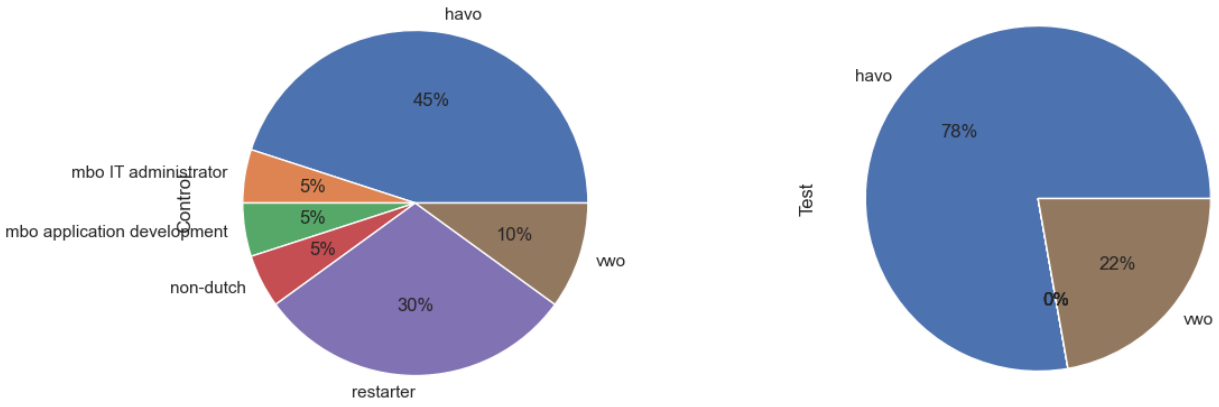


Figure 4.3: Previous education split by group for students with correct implementations

One would expect students with previous programming experience to do better on a programming assignment. On the other hand, this expectation could be wrong for programming assignments worded at the level of higher education. The comprehension level of those students with havo and vwo as previous education could be higher, making it easier to come up with a better solution. Figure 4.4 shows students redoing the course have higher marks. These students are not subdivided by earlier education, and have an advantage in their experience with the course. As such, we disregard their higher scores. Students with education outside of the Netherlands are also disregarded, as well as the students who took the 21+ test. Both of these groups overlap with the other groups in terms of educational level, but foreign educational systems are not easily compared to the Dutch system, and the 21+ test only says something about the fitness for a specific program, not about an absolute level. We did see students from mbo scoring lower than both havo and vwo, and since the test group had no students who have mbo as previous education, it might have been at an advantage.

4.3.1. Conclusion

In conclusion, how is the quality of students' work affected by this education? We do see a clear difference between the test and control groups. However, are the differences we see directly traceable to our work? To answer that question, repetition of our research with larger student groups is necessary, to increase the confidence in the data. In our current setup, one student in a pair may make a lot of difference in the resulting mark. The recorded pair programming sessions show no difference in behavior, and there does not seem to be an indication in the interviews that students took a different approach because of the additional training. One explanation could be the larger share of havo and vwo students the test group had. In the next section, we will see if the same differences can be seen when looking at the individual students' mark on the course exam.

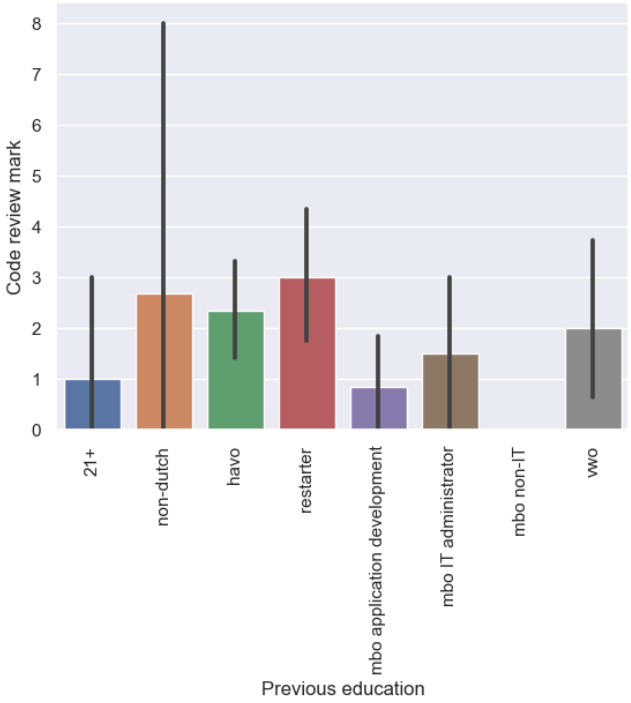


Figure 4.4: External code quality for weekly assignment split by previous education

4.4. EXAM RESULTS

In this section, we answer the question: What is the difference in exam scores between students with and without this education? Our data to answer this question included 108 individual exam marks, 34 from the test group, 74 from the control group, for which we will see if a notable difference can be seen between students who did get instruction in specification writing and those that didn't. The exam consisted of five small programming assignments that test comprehension of basic programming skills, such as variables, control flow, classes and methods. Specification writing was not part of the exam.

First, we examine the average test result for the test and control groups in figure 4.5. As the control group is larger, the confidence interval is tighter, but we see a nearly identical average.

In the previous section, we saw a big difference in external code quality that might be related to previous education of the participants. When accounting for this finding by considering just those students with havo as previous education, we see that the average for the two groups is slightly lower than the whole control group, but that the difference is negligible (test group: 7.0, control group: 6.9). The amount of practice both groups had in the MOOC was exactly the same, with an average completion rate of 68%. Interestingly, when looking at just the vwo students, we see a slight difference between the groups (test group: 9.3, control group: 9), but especially the difference of two points with the havo group is stunning. For the control group, the average MOOC completion rate of 75% is slightly higher than for both the havo groups, but for the test group the average completion rate is much higher: 92%. This might account for the difference in average score.

As the control group students had different instructors guiding them while practicing with the MOOC, we show the same results per instructor group, as seen in figure 4.6. The A and B groups form the control group, and the C group is the test group.

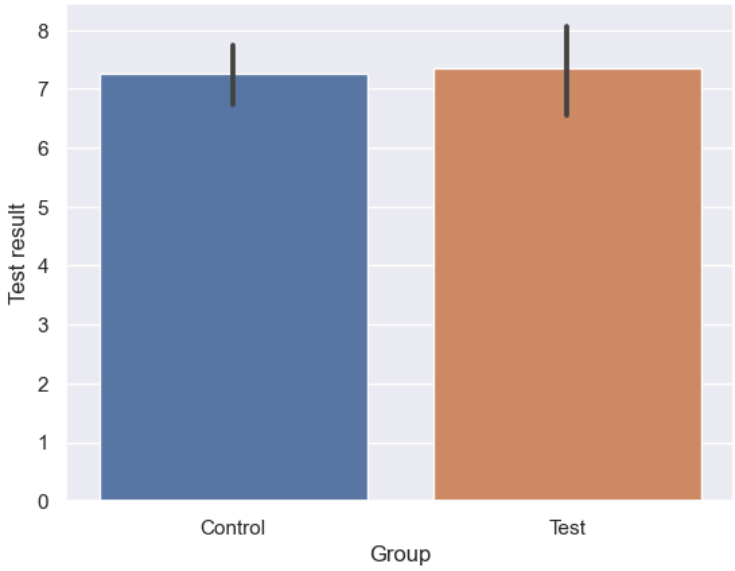


Figure 4.5: Average test result by group

A difference of 1.4 points can be seen between the A and B groups. Since both groups were instructed with the same topics and all students considered had havo education, the way the instructor guided his students while practicing with the MOOC might have an impact on the test result. This impact can come from extra tips they gave, but not from grading, as the test were graded horizontally, which means instructor A grades question 1 for all students, instructor B grades question 2 for all students, and so on. Nor can it come from the general instruction, as this was given simultaneously to all students. As shown in figure 4.7, the distribution of chosen profiles of the havo students in groups A and B is nearly identical, indicating this is not a factor in the difference between both groups.

4.4.1. Conclusion

In conclusion, what is the difference in exam scores between students with and without extra education in specification writing? On the whole, no difference is seen between the test and control groups. Only when separating the control group per instructor, we see a difference in the average grade. At first glance, this seems related to the instructor guidance, but could very well be influenced by underlying factors that have traditionally been linked to success in learning programming, such as self-efficacy [Fincher et al., 2006][Watson et al., 2014]. The gap between havo and vwo students is also very large, independent of instructor. It is worth investigating whether all vwo students are better equipped to learn programming at the start of the the program, and if so, why this is. In our next chapter, we will discuss if the results from this and previous sections have scientific meaning and can lead to further research.

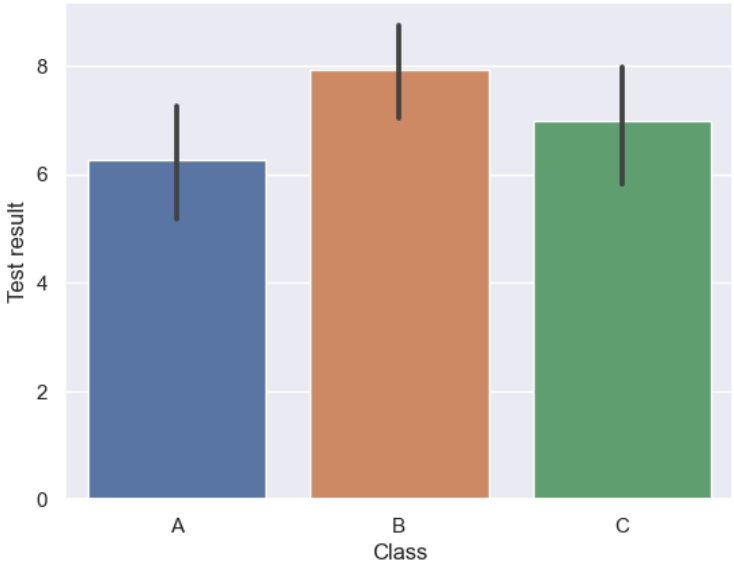


Figure 4.6: Average test result by class for students with havo as previous education

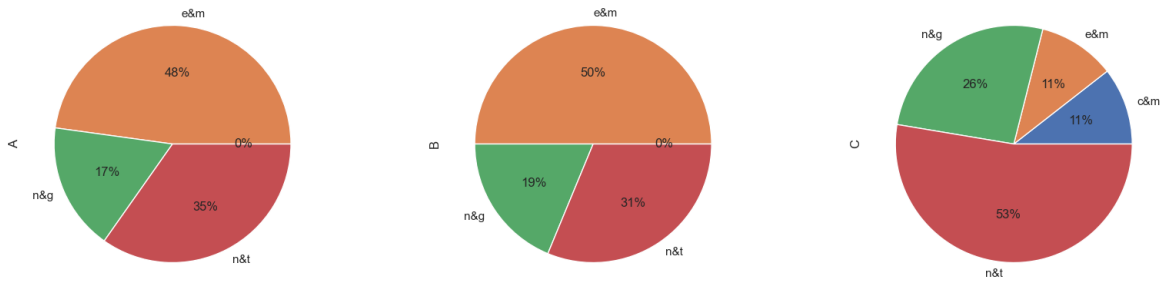


Figure 4.7: Composition of classes by profile chosen in previous education, for havo students (**C**ultuur & **M**aatschappij), **E**conomie & **M**aatschappij, **N**atuur & **G**ezondheid, **N**atuur & **T**echniek)

5

Discussion

The previous chapter showed our results and some of the tentative conclusions we drew. In this chapter, we will interpret our results in cohesion, discuss their scientific value and recommend future work.

The material created to teach students specification writing is considered of sufficient quality by students and was received well by our academic peers, but students for the most part have not chosen to use the method, at least to some extent because they do not see the value in documenting code. Students at least seem to understand the importance of analyzing an assignment, as shown by their efforts towards analysis for the algorithmic approach. But alas, it seems students are fast to succumb to the fallacy that topics not graded are not worth learning [Kohn, 1999] and as such, did not practice with our method.

It is vitally important students practice specification writing before measuring its benefits. Because students did not choose to do this, our research setup essentially failed. When not exercising, the offered knowledge is not persisted, without which any conclusions towards specification writing making a difference are debatable at best. The results we present should be interpreted as such, and can merely be used to plan for further research, for which we have multiple recommendations. Most importantly, provide an incentive for students to participate in the experiment, so that the intervention can actually be evaluated. Consider starting with just reading specifications, before asking students to write specifications. Finally, ensure no or minimal difference between the control and test groups other than the specific intervention measured. We explain these recommendations in more detail below.

In order to move students to practice specification writing, an incentive is needed. We did not choose to implement this in our research, as we had the intention of comparing groups in the same year cohort. We did not expect first year students to understand the importance of properly documented code, because they lack the adverse experience of trying to understand code written by another programmer, or even their past self, but we hoped students at the start of their study would still be compliant enough to accept its importance when told as much, and use the method. If one were to award points to incentivize practice with our method however, they would lose the possibility of comparing control and test groups in the same year cohort, as it would be hard to justify grading two groups from the same cohort differently in the same educational setting. A way to circumvent this would be to treat two different cohorts as the test and control group, while assuring no other factors

change between the years. In our setting, this seems difficult, as the course changes each year based on student feedback, experiments and the instructors coupled to it.

In all further research with our method, we do recommend tying in some incentive to practicing with it. This could be as simple as adapting the automated tests of the MOOC to also test for the presence of complete specifications. Students seem to be sensitive to a red cross turning into a green tick. Another way could be to have students complete or continue with work that is not their own, so that they may experience the benefit of specifications for themselves. When choosing an award in credit, the simplest approach is to have the topic of specification writing count towards the exam mark.

It needs to be mentioned our study took place in the middle of the COVID-19 pandemic. Students studied from home for the most part and were out of sight for the instructors for the most part. Their progress in the MOOC was easily tracked, but where normally an instructor can steer students in the class room when they get sidetracked, this was not the case with online-only education. Normally, the instructor would be able to steer the students towards practice with our material, but as this was not the case now, the effectiveness of guidance was diminished. Perhaps in a class room setting, instructors would have been able to coax students into practicing with our material.

We do consider our material to be of high enough quality to reuse in different settings. Specification writing is an essential tool in writing better tests [Leventhal et al., 1994], which alone is reason enough to introduce it to aspiring software engineers. However, we have some concerns on the place of introduction. For starters, we attempted to teach specification writing without immersing the student in the use of specifications first. Perhaps the better choice would be to start them off with specification use, rather than asking them to write them. If students are used to having specifications available, the step towards recognizing their usefulness would be a lot smaller. Not having beginning students write specifications is in line with the conclusions of Buck and Stucki [2000]. Furthermore, letting students get used to specifications before asking them to write them will possibly make them acquainted with specifications in such a way that in their absence, they miss them, and see their benefit without knowing the intended outcomes of specifications. In this way it is possible to circumvent the issue where students don't apply our method when they do not see the benefit yet.

Our expectation at the start of our research was that having students write specifications would make them write better code, merely because the act of thinking about the purpose and possible failure modes of their code, would trigger them to think about what they could to fulfill that purpose robustly. Although we did see a clear difference in external code quality between the test and control groups, it is not possible to directly relate this to the act of specification writing, as students chose not to do this. Mere instruction in specification writing is unlikely to influence their mindset in such a way that they naturally write better code. In further research with our method, it would be best to choose the test group in such a way that unintended variables, such as the approach of an instructor, previous education, or the composition of their study groups, do not influence the outcome, so that a difference in quality can be correlated with specification writing.

The quality of code was determined through code review for one assignment made by pair programming. This endeavour was recorded for later analysis, so that the thought process could be seen, as well as the end result. In the recordings it was seen that most pairs were very eager to start coding. In general, students did not seem confident in their ability

to solve the assignment, and were happy to have finished it. The assignment was the last in a series of weekly assignments, which could lead students to believe it was the hardest, and exhibit the lack of confidence we saw. During coding, it was apparent some groups lacked even the most basic of programming skills, with one group in particular taking 45 minutes to find out how to create an array, a skill they should have learned three weeks prior. While this was an extreme, the average skill shown was surprisingly low. As the students were happy just to have working code, it seems that a quality mindset can only grow when they become more confident in their ability. It is probably best to introduce specification writing at a later point in their education, when they are ready to move past just getting code to work. Students at these early stages of their education will probably benefit more from (automated) code reviews. Students at these stages may also benefit from instruction in going from a problem statement to an algorithm, for which some students now had formed their own strategies, and instruction in basic debugging. Student confidence in their approach may benefit greatly from tools with which they can tackle any problem in a structured manner, and as a result, their success in the introductory programming course may be higher.

We did not have a hand in the forming of the pairs for the pair programming exercise, which led to some interesting combinations. A lot of groups had one very vocal student and one more quiet one. This was not due to the classic driver-navigator division, but rather seemed to be coupled to each students skill level. In some pairs, where one students skill was significantly higher than the other's, the student with the higher skill level did all the work. We did not benefit from the thinking aloud method in those cases. In future work, it might be prudent to form pairs of somewhat equal skill when applying this method. The choice of not observing the students directly turned out to be the right one. Although students showed themselves aware of a recording being made, on multiple occasions they displayed surprise when it was revealed the recordings were actually analyzed. This seems to show they did not feel observed when pair programming, so that the change in their behavior due to being observed was kept to a minimum.

We saw a 13% difference in correctness of the output of the peer programming sessions, in favor of the control group. As all external code quality attributes we scored on were higher for the test group, this one surprised us. We have no evidence that the test group spent more time on analysis, but perhaps some aspects of our training influenced them in such a way that they spent less time on implementation and checking their implementation. It is entirely possible the influence of the control group instructors is again at play here, such that they focused more on testing code in their guidance, while the test group instructor spent more time coaching on analysis. Again, if an instructor has this much influence on student performance, in follow-up research it would be best to divide the test group between all instructors.

Although we did see a significant difference in external code quality during code reviews, it should be noted the code reviews were done by the instructor for the test group, which could mean the difference comes from him noticing preferred patterns in the code, he taught to that student group himself. In that sense, perhaps the exam results are a better indicator for the effect training in specification writing has on programming ability. It was no surprise the test and control groups scored similarly, as the test group students did not practice with our method, but the large gap between the groups that had different instructors in the control group was. Even when looking at just havo as previous education, with almost the exact same composition of profile choices, we saw this gap. The interaction

time students had with their specific instructor was low, at most half an hour per week. It would be very interesting to do follow-up research on if a difference in grading scores this large can be caused by the influence an instructor has with so little time, and what specific approach leads to students performing better. Our hypotheses is this instructor puts emphasis on a bond with his students, which motivates them to put in more effort. In follow-up research it would be best to divide the test group between all instructors, and to ensure the compositions of the test group and control group, by previous education, are similar.

Even more surprising was the gap between havo and vwo students. Vwo students account for 14 out of 108 students, including two students who started the program anew in their first year, but for these 14, the average exam score was two points higher than the average score for havo students. It would be very interesting to follow-up this accidental finding, to see what the difference between havo and vwo students are before starting the program, and to investigate whether starting havo students could, in some way, benefit from extra preparation in the program. Our hypothesis is that vwo students are more skilled at abstract thinking than havo students. Additionally, it seems prudent to investigate whether havo students benefit from a vwo student in their ranks, and if the difference in ability persists throughout the program.

With the influence of previous education seemingly of significance, it is worth mentioning the test group was formed by one of the classes that were semi-randomly assigned to at the beginning of the propaedeutic year. Two factors played a role in how students are assigned to a class. Each student has the option to request being placed with another student they know, and women are distributed among the study groups evenly, because, although we have only anecdotal evidence to support this, they seem to have a positive influence on the study behavior of the group as a whole. Otherwise, the classes are put together randomly and one of the classes became the test group. This does leave the possibility of the control and test groups having a completely different composition, which in turn can lead to an unintended variable.

6

Conclusion

In our research, we attempted to answer in what ways students benefit from education in writing informal specifications. To this end, we used four sub-questions. In conclusion, we will summarize the answers found to these questions and answer the main question.

What does effective education in specification writing look like? We created quality material to teach specification writing based on four-component instructional design. The material can be used in a higher education setting, which we did with a sub-group of students we call the test group. Although most students indicate they saw benefit in our instruction, they did not choose to apply our procedural guidance during exercises, probably because they do not yet see the benefit of writing specifications on the whole, and we gave them insufficient incentive to do so. Effective education in specification writing applies our material, but also adopts an approach in which students are appropriately incentivized to practice specification writing.

What difference can be seen in students' approach to problem-solving with and without this education? As students did not practice with our material, and did not adopt if for the assignment we gave them to analyze their approach on, we did not see an actual difference between our test and control groups. In general, students seemed happy enough and relieved that they reach the goal of the assignment, regardless of how this goal was reached. Although it was not the intention of our research, we did notice our students exhibit the same kind of behavior Kolikant described [Kolikant, 2005].

How is the quality of students' work affected by this education as compared to the work of students without this education? We saw a clear difference between the test and control groups, which we can't link to our research just yet. The test group scored significantly higher on external code quality than the control group. As our sample size is small, repetition of the research would be needed to see if the difference stems from our instruction, or another factor. The assignment used for this needs slight adjustments, but can be reused in follow-up research when working with a different student group.

What is the difference in exam scores between students with and without this education? No difference is seen between the test and control groups. We did see a large difference between two instructor groups in the control group. This was not a difference we were looking for, but its an interesting finding nonetheless. Compared to the lower scoring group of the two, the test group does better, but does worse when compared to the higher scoring group. We also saw a large difference in test scores between havo and vwo students, with vwo students scoring a full two points higher on average.

In what ways do students benefit from education in writing informal specifications? Taking into account the answer on our sub-questions, we have to conclude the main research question remains unanswered. The research was set up in such a way that student participation was not coerced and student awareness of the study did not taint the results. This has inadvertently led to students not practicing specification writing and, although we did see a difference in external code quality, we could not link this to our method directly. Still, our research adds to the body of knowledge of software engineering by providing material for training on specification writing, with the accompanying procedural guidance being useful for both educational and professional use. We are still confident that, if deployed correctly, the writing of specifications will positively impact code quality. Although our research question remains unanswered, we gained valuable insight in how to better perform our research.

6.1. Future work

Repeating this research is a logical possibility for future work. When doing so, we recommend using some incentive to promote practice, be it extra credit or a more intrinsic approach, such as using automated tests to check for specifications. Larger test and control groups are needed to increase the confidence in the findings. Also, it would be prudent to divide the test group among instructors to prevent unintended factors influencing the research. The effect an instructor has on student test results, through their approach in instruction, also makes for interesting future work.

One of the benefits of specification writing was to remedy students only writing happy path test cases. With our material, it is possible to investigate whether better tests are actually written by students who follow it. When doing so, we recommend first letting students getting used to working with specifications by providing them, rather than letting the students write specifications themselves.

Lastly, the difference between the performance of havo and vwo students on the final exam should be investigated to see what attributes vwo students have that makes them better equipped for learning programming, and if these attributes can be leveraged to benefit havo students.

BIBLIOGRAPHY

A. Bijlsma, N. Doorn, H. Passier, H. Pootjes, and S. Stuurman. How do students test software units? In *JSEET - Joint Track on Software Engineering Education and Training of ICSE, 43rd International Conference on Software Engineering*, page to appear, 2021. 1

Duane Buck and David J. Stucki. Design early considered harmful: Graduated exposure to complexity and structure based on levels of cognitive development. In *Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '00, page 75–79, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581132131. URL https://doi.org/10.1145/330908.331817. 1, 6, 27

Michael J Clancy and Marcia C Linn. Patterns and pedagogy. *ACM SIGCSE Bulletin*, 31(1): 37–42, 1999. 4

Stephen H. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. *SIGCSE Bull.*, 36(1):26–30, March 2004. ISSN 0097-8418. URL https://doi.org/10.1145/1028174.971312. 1, 4, 11

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. URL https://doi.org/10.1145/2591708.2591757. 1, 6

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing*. The MIT Press, 2018. ISBN 0262534800. 5

Sally Fincher, Anthony Robins, Bob Baker, Quintin Cutts, Patricia Haden, Margaret Hamilton, Marian Petre, Denise Tolhurst, Ilona Box, Michael de Raadt, et al. Predictors of success in a first programming course. In *Conferences in Research and Practice in Information Technology Series*, 2006. 24

Richard Herbert Franke and James D Kaul. The hawthorne experiments: First statistical interpretation. *American sociological review*, pages 623–643, 1978. 12

Jimmy Frerejean, Jeroen JG van Merriënboer, Paul A Kirschner, Ann Roex, Bert Aertgeerts, and Marco Marcellis. Designing instruction for complex learning: 4c/id in higher education. *European Journal of Education*, 54(4):513–524, 2019. 9

Paul Kirschner and Jeroen Van Merriënboer. *Ten steps to complex learning a new approach to instruction and instructional design*. Taylor & Francis Ltd, 2008. 3

Alfie Kohn. *Punished by Rewards: The Trouble with Gold Stars, Incentive Plans, A's, Praise, and Other Bribes*. Houghton Mifflin Harcourt, 1999. 26

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. URL https://doi.org/10.1145/1089786.1089790. 1, 4, 17, 30

Laura Marie Leventhal, Barbee Eve Teasley, and Diane Schertler Rohlman. Analyses of factors related to positive test bias in software testing. *Int. J. Hum.-Comput. Stud.*, 41 (5):717–749, November 1994. ISSN 1071-5819. URL https://doi.org/10.1006/ijhc.1994.1079. 1, 27

Bertrand Meyer. Applying'design by contract'. *Computer*, 25(10):40–51, 1992. 5

Mitchell J Nathan, Kenneth R Koedinger, Martha W Alibali, et al. Expert blind spot: When content knowledge eclipses pedagogical content knowledge. In *Proceedings of the third international conference on cognitive science*, volume 644648, 2001. 1

Jonathan S Ostroff, David Makalsky, and Richard F Paige. Agile specification-driven development. In *International Conference on Extreme Programming and Agile Processes in Software Engineering*, pages 104–112. Springer, 2004. 6

George Polya. *How to solve it*. Princeton university press, 1957. 4

R Douglas Riecken, Jurgen Koenemann-Belliveau, and Scott P Robertson. What do expert programmers communicate by means of descriptive commenting. In *Empirical studies of programmers: Fourth workshop*, pages 177–195. Ablex, 1991. 16

Simon Thompson. Where do i begin? a problem solving approach in teaching functional programming. In *International Symposium on Programming Language Implementation and Logic Programming*, pages 323–334. Springer, 1997. 4

Jeroen JG Van Merriënboer. *Training complex cognitive skills: A four-component instructional design model for technical training*. Educational Technology, 1997. 8

Jeroen JG Van Merriënboer and Paul A Kirschner. *Ten steps to complex learning: A systematic approach to four-component instructional design*. Routledge, 2017. 3

MW Van Someren, YF Barnard, and JAC Sandberg. The think aloud method: a practical approach to modelling cognitive. *London: AcademicPress*, 1994. 11

Christopher Watson, Frederick WB Li, and Jamie L Godwin. No tests required: comparing traditional and dynamic predictors of programming success. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 469–474, 2014. 24

Leon E Winslow. Programming pedagogy—a psychological overview. *ACM Sigcse Bulletin*, 28(3):17–22, 1996. 1, 4

APPENDIX A: SUITABLE EXERCISES FROM THE JAVA PROGRAMMING MOOC

These exercises are part of the Java Programming MOOC found at java-programming.mooc.fi. For each of the parts in the sidebar, find the list of exercises in this part. The exercises named below are prefixed by 'Programming exercise'.

PART 3

- Print in range
- Sum
- Remove last
- Sum of array
- Print neatly
- Print in stars

PART 4

- Whistle
- Door
- Debt
- Song
- Film
- Gauge
- Multiplier
- Payment Card

PART 5

- Fitbyte
- Comparing apartments
- Santa's Workshop
- Longest in collection
- Height Order
- Cargo hold

APPENDIX B: PROGRAMMEREN 1 WEEKTAAK 5 (IN DUTCH)

In de medische wereld wordt gebruik gemaakt van een Elektronisch Patiënt Dossier waarin voor een patiënt informatie rondom gezondheid bijgehouden wordt. Je maakt de classes die voor een EPD van belang zijn. De eisen hiervoor zijn:

- Het is mogelijk om een arts aan te maken.
  - Een arts heeft een naam en een specialisme.
- Het is mogelijk om een onderzoek aan te maken.
  - Een onderzoek heeft een naam, een uitslag en een specialisme.
  - Een onderzoek heeft (wordt uitgevoerd) door een arts.
  - Het specialisme van de arts moet overeenkomen met het specialisme van het onderzoek.
- Het is mogelijk om een patiënt aan te maken.
  - Een patiënt heeft een naam, geboortedatum, geslacht en BSN.
  - Een patiënt kan meerdere onderzoeken gekoppeld hebben.
- Het is mogelijk om het EPD op het scherm te tonen. Voor een patiënt wordt getoond:
  - Naam, geboortedatum, geslacht en BSN.
  - Alle gekoppelde onderzoeken, met voor elk onderzoek:
    - Specialisme, naam, uitslag en arts.

Het is niet verplicht om een complete user interface aan te leveren, maar laat zien dat je in ieder geval aan de eisen voldoet door het gebruik van je classes in de main methode van je applicatie.

APPENDIX C: PROGRAMMEREN 1 WEEKTAAK 6 (IN DUTCH)

WEERSTANDEN

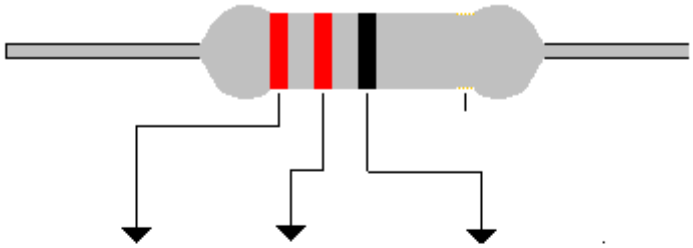
Een weerstand is een elektronische component dat in bijna alle elektronische apparatuur voorkomt. De waarde van zo'n weerstand wordt uitgedrukt in Ohm naar de Duitse wetenschapper Georg Ohm. Het symbool dat voor weerstand gebruikt wordt is Ω. Weerstanden worden meestal geproduceerd volgens de E-12 reeks. De mogelijke waarden van die reeks zijn 1.0, 1.2, 1.5, 1.8, 2.2, 2.7, 3.3, 3.9, 4.7, 5.6, 6.8 en 8.2, eventueel vermenigvuldigd met machten van 10. De vermenigvuldiging noemen we de multiplier. In de tabel is dat aangegeven met MULTIPL. Veel gebruikt worden de termen k (kilo) voor 1.000 en M (mega) voor 1.000.000. Zo kunnen we bijvoorbeeld een weerstand hebben van 68 kΩ = 68.000 Ω (achtenzestig kilo Ohm).

Om de waarde van een weerstand te kunnen aangeven op de weerstand zelf, wordt een kleurcode gebruikt in de vorm van drie ringen. Elke ring heeft een waarde, die gezamenlijk de weerstandswaarde uitdrukken. Dit is te zien in de tabel zoals te zien in figuur 1. De eerste en tweede ring geven aan om welke waarde uit de E-12 reeks het gaat. Rood-rood geeft bijvoorbeeld aan dat de E-12 waarde 2.2 is. De derde ring geeft aan met hoeveel die waarde vermenigvuldigd wordt. Een zwarte ring geeft bijvoorbeeld een multiplier van 1 aan, waardoor de weerstandswaarde 2.2 Ω is.

Een tweede voorbeeld: Stel, we willen een weerstand van 4100 Ω. De dichtstbijzijnde waarde uit de E12 reeks is 3.9 met een factor van 1000. We krijgen dan een berekende waarde van 3900 Ω. Precies 4100 Ω zal met één weerstand niet gaan. De kleurcode voor deze weerstand is oranje, wit, oranje: oranje = 3, wit = 9 en oranje als derde betekent een factor 1000. Dit levert samen een weerstand van 3900 Ω. De dichtstbijzijnde optie naar boven is 4.7 met een factor van 1000. Deze combinatie levert een weerstand van 4700 Ω. Omdat 3900 Ω dichterbij ligt, nemen we die waarde als uitkomst.

OPDRACHT

Schrijf een methode die voor een gegeven weerstandswaarde de dichtstbijzijnde mogelijke weerstand uit de E-12 reeks geeft. De methode moet zowel de waarde van de geschikte weerstand als de kleurcode teruggeven. Hoe je de methode vormgeeft staat je verder vrij, als deze maar een input heeft en met een return waarde werkt. Het gaat hier om een complexe opgave, dus besteed voldoende tijd aan het analyseren van wat er precies gevraagd wordt voor je de code induikt.



KLEUR	1e RING	2e RING	MULTIPL.
ZWART	0	0	1
BRUIN	1	1	10
ROOD	2	2	100
ORANJE	3	3	1k
GEEL	4	4	10k
GROEN	5	5	100k
BLAUW	6	6	1M
VIOLET	7	7	10M
GRIJS	8	8	
WIT	9	9	

Figure 1: Tabel met kleuren en factor voor weerstandswaarde volgens E-12

APPENDIX D: PROCEDURAL GUIDANCE FOR WRITING INFORMAL SPECIFICATIONS FOR PUBLIC METHODS

Note: In each step you may encounter incomplete information. In a business environment this is a point at which you will request more information from the relevant stakeholders. In school, you can often ask the teacher for this information, or make an assumption. It is important to document the latter, because if the assumption no longer holds, your specification might have to be adapted.

STEP 1: READ THE ASSIGNMENT IN ITS ENTIRETY.

Get a feel of the problem statement. Do not delve into the details just yet. When progressing to the next steps, it helps to have an overview of what is asked of you. Pay attention to the order in which the problem statement is written. For example, you will often find restrictions on input described in a different section than what the method should do with that input.

Note: As you grow as a software engineer, you will get assignments for larger programs that you will create a design for. Part of the design is dividing the program in methods. This is called *decomposition*. For each decomposed method you will follow these steps. In Programming 1, you are always provided with the decomposition in the assignment.

STEP 2: SUMMARIZE THE PURPOSE OF THE METHOD.

Now that you have read the problem statement, you can make a summary of what the method is supposed to do. Start the summary with a verb to keep it concise. Try to write the summary in such a way that it can be understood without the rest of the specification. Be specific when describing a term that may lead to ambiguity, convey what you understand the term to mean. If this would take too much text, you can link to a source.

Bad example: This method takes a double x and multiplies it by itself.
Good example: Returns the square of the given number.

STEP 3: DEFINE THE PARAMETERS OF THE METHOD.

If a method needs input for its purpose, this is usually defined in the method parameters. If no input is needed, or the input is acquired within the method in some way, you may skip this step. If one or more method parameters are needed, take these steps for each one:

a. Define its purpose. Defining the purpose of a parameter helps filling in the blanks when your client is confronted with incomplete information, and helps with naming the parameter and determining its type. Be specific when describing a term that may lead to ambiguity, convey what you understand the term to mean.

b. Name the parameter. In Java, each parameter has a name so that it can be referred to. For this purpose, the actual name is not important, but for human readers a name makes all the difference in the world. Use step 3a to think of a name that instantly communicates the purpose of the parameter.

Bad example: Naming a number to square 'x'.
Good example: Naming a number to square 'root'.

c. Determine the type of the input. In Java, each parameter has a type that determines what values can be given as the input argument. Determine the type by what is appropriate for your input. For example, truth values are represented with a Boolean. Numbers are represented with number types, such as int, float or double. It is possible to represent both of these with a String, but this makes your implementation unnecessarily complex.

Note: As you progress as a programmer, you will learn more about the many intricacies to choosing the right parameters. For now, these will often be given in the assignment.

d. Find out what the restrictions on your input are. When you know the type of your parameter, you know all the possible values that parameter can take. However, not all possible values are appropriate for your method. For example, if your method has an int parameter, is a negative value appropriate? Pay special attention to edge cases and *null*. If the assignment says a number from 0 to 10 must be given as input, is 10 included? If the input is a String, can any character be used?

STEP 4: DETERMINE DEPENDENCIES ON THE STATE OF YOUR PROGRAM, AND BETWEEN INPUT PARAMETERS.

In the previous step you determined the restrictions on every input parameter. There might be interdependencies between the parameters as well, meaning that the validity of one value may be dependent on the value of a different parameter. For example, if a method has two parameters, an integer indicating the number of eyes on a dice and an integer with the last roll of that dice, the second cannot exceed the value of the first, Write down such interdependencies.

Secondly, calling the method might be dependent on the state your program is in. For example, if a different method needs to be called before this one, specify this.

STEP 5: DEFINE THE OUTPUT OF THE METHOD.

All methods are intended to produce a certain output and/or side effect. For the intended output:

a. Define the purpose. Defining the purpose the output helps filling in the blanks when your client is confronted with incomplete information, and helps with determining a return type. Be specific when describing a term that may lead to ambiguity, convey what you understand the term to mean.

b.Determine the return type of the method. If the output of the method includes a return value, determine the type by what is appropriate for your output purpose. If multiple

return values are needed, consider a composite type. If no return value exists, the type will always be *void*.

c. Think of the different output specific input will give. A method might give different output for different inputs. A positive number might lead to a return value of the square of that number. A negative value when a positive value is expected can cause an exception to be generated. A String that is not found in a database can cause Boolean value *false* to be returned. Define the output for input that adheres to the restrictions from step 3d and 4, but also determine what should happen in the inverse situation, when the input does not comply with the restrictions. In Java, there is no need to check the type of the given input, so do not describe wrong input type scenario's. It is very helpful for implementers and testers to think of examples of input/output combinations at this point.

STEP 6: NAME THE METHOD.

Use a name that instantly makes clear what the purpose is. Do not repeat the input names or purposes.

Bad example: Naming a method determining the square root of a value 'calculateSquareRootIntFromIntValue'.
Good example: Naming a method determining the square root of a value 'squareRoot'.

STEP 7: BUILD THE SPECIFICATION.

The specification should use the result of the previous steps to specify the method.

```
/**
* <summary from 2>
*
* @param <name from 3b> <paraphrasing of purpose from 3a>
* @return <paraphrasing of purpose from 5a, if a return value exists>
*
* @requires <restrictions on input from 3d and 4>
* @ensures <outputs for different inputs from 5c>
* @also
* @requires <inverse of restrictions on input from 3d>
* @throws <exception to throw, usually AssertionError in Programming 1>
*/
public <type from 5b> <name from 6>(<type from 3c> <name from 3b>){...}
```

Note that multiple sets of @requires and @ensures can exist, as well as multiple sets of @requires and @throws. When multiple parameters are used, the specification contains as many @param tags.

THE GUIDANCE IN USE (NOT PROVIDED TO STUDENTS)

Specification for 'Weerstanden' assignment, see Appendix C. Step 1 is reading the assignment, we will start from step 2.

STEP 2

Returns the closest appropriate E-12 resistor value and its color code for a given desired resistance value.

STEP 3

One parameter (given), which is a resistance value:
a. The resistance value is a rational number, expressed in Ohm. The maximum resistance value possible for a single resistor is in the millions of Ohm's. The resistance value given is a desired resistance value, not necessarily a possible one in the E-12 range.
b. desiredResistance
c. As the maximum resistance for a single resistor is in the millions and is a rational number, the Java type double will be used.
d. A double accepts negative values and zero as input. These are not valid resistance values. We treat these the same as input 1.0. Values that are higher than the highest E-12 value (8.2KOhm) are treated as input 8200000.0.

STEP 4

There is one input parameter. Calling the method does not depend on the state of the application.

STEP 5

a. The output is both the precise value as available in the E-12 system and the color code representing this value, consisting of three colors.
b. Multiple return values are needed, this will be a composite type Resistor. Resistor is not specified here, but is an object encapsulating Enum values for the colors, and a double resistance value.
c. desiredResistance < 1.0 returns new Resistor(Color.BROWN, Color.BLACK, Color.BLACK, 1.0) desiredResistance > 8200000.0 returns new Resistor(Color.GREY, Color.RED, Color.VIOLET, 8200000.0) desiredResistance >= 1.0 and desiredResistance >= 8200000.0 returns a calculated resistance value and color code.

STEP 6

chooseE12Resistor

STEP 7

```
/**
* Returns the closest appropriate <a href=https://en.wikipedia.org/wiki/E_series_of_pre
* E-12 resistor value</a> and its <a href=https://en.wikipedia.org/wiki/Electronic_colo
* color code</a> for a given desired resistance value.
*
* @param desiredResistance Rational number, expressed in Ohm.
* @return Resistor object with precise E12 value and color code.
*
* @requires desiredResistance < 1.0
* @ensures new Resistor(Color.BROWN, Color.BLACK, Color.BLACK, 1.0)
* @also
* @requires desiredResistance > 8200000.0
* @ensures new Resistor(Color.GREY, Color.RED, Color.VIOLET, 8200000.0)
```

```
* @also
* @ requires desiredResistance >= 1.0 && desiredResistance <= 8200000.0
* @ ensures Resistor with E12 resistance value and color code calculated from input
*/
public Resistor chooseE12Resistor(double desiredResistance) {}
```

Appendix E: Survey effectiveness of addition to Programming 1 course (in Dutch)

Analyse en specificatie

Dit jaar hebben we extra instructie over analyse en specificatie uitgeprobeerd. Deze vragen zijn specifiek op dit onderdeel. Hoe uitgebreider je antwoord, hoe beter we onze vakken kunnen maken!

Geef per stelling weer aan in hoeverre je het ermee eens bent. Kies van 1 (helemaal niet mee eens) t/m 5 (helemaal mee eens).

1. Ik vond 'analyse en specificatie' een interessant onderdeel. **1 2 3 4 5**

2. De kwaliteit van de lesmaterialen op Blackboard voor 'analyse en specificatie' is hoog. **1 2 3 4 5**

3. De effectiviteit van de training 'analyse en specificatie' was hoog. **1 2 3 4 5**

4. Het was tijdens de training 'analyse en specificatie' duidelijk hoe het onderwerp terugkomt als je al programmeur in een bedrijf werkt. **1 2 3 4 5**

Ja/Nee

5. Heb je tijdens het maken van opdrachten gebruik gemaakt van het stappenplan voor analyse en specificatie (procedural guidance) **Ja (ga naar 6a) / Nee (ga naar 6b)**

Open vragen

6a. Helpt het gebruik van het stappenplan in hoe je een opdracht benadert en op welke manier?

6b. Wat is de reden dat je niet gekozen hebt om het stappenplan te gebruiken?

7. Op welke manieren heb je baat gehad bij de extra instructie?

8. Wat wil je niet veranderen in de extra instructie en de lesmaterialen?

9. Wat zou je veranderd willen zien aan de extra instructie en de lesmaterialen?

10. Dit wil ik nog kwijt over de extra instructie:

Appendix F: Assessment model for 'Weerstanden' assignment

Item	Criterion	Quality	Points
Method	Recognizes the assignment asks for a method.	External	2
Method name	Method name shows the method returns a valid resistance for a desired value.	External	1
Comments	Specified the method with Javadoc.	External	1
Guards	Checks if input arguments are in a valid range.	External	1
Output	Uses a class to return a composed value.	External	2
Input	Input argument type is double.	External	2
Input name	Parameter name shows the input is a resistance value.	External	1
Correct	Code works correctly for examples given in assignment.	External	3
InternComm	Code is commented within the method.	Internal	1
! E12	The code can easily be adapted to other ranges than the E12 variant.	Internal	1
Colors in array	Recognizes the color table values are easily translated to array indices.	Internal	1
Multiplier	Recognizes multiplier value can be translated to array indices using log.	Internal	1
Calculated	Values are calculated instead of using string manipulation.	Internal	1