UiO : Department of Informatics
University of Oslo

Errors and misunderstandings among novice programmers

Assessing the student not the program

Mathias Johan Johansen

Master's Thesis Autumn 2015

Errors and misunderstandings among novice programmers

Mathias Johan Johansen

November 2, 2015

Abstract

Novice programmers make a lot of programming errors as they strive to become experts. This is a known fact to teaching faculty in introductory programming courses. The errors play a major role in both formative and summative assessment of the students. The Computer Science research of today trends towards focusing on automatic assessment of program, becoming more remote from the student who wrote the program.

In an attempt to create a better understanding of the novice programmers and the errors they make, this thesis takes a look at the errors novice programmers produce in programming assignments and the misunderstandings that may have caused them.

We use a qualitative approach to analyze assignments, interviews and observations with 23 students of a second semester course in object-oriented programming, using Java, at the University of Oslo. 33 solutions to mandatory assignments were analyzed to identify errors. 14 students were interviewed about their solutions to identify the misunderstandings that caused the errors we identified. Nine students participated in think-aloud observations to add further insights into how students of the targeted course approach problem solving. Finally the misunderstandings are analyzed to review the feasibility of using misunderstandings to guide formative and summative assessment.

We have identified multiple student errors in both result and design. Multiple errors and misunderstandings revolved around generic class parameters, a topic that not much existing research covers.

Our findings suggest that misunderstandings considering the principles of object-oriented design lead to further misunderstandings about the important aspects that must be considered to write good programs.

We did not find a feasible way to use pre-identified misunderstandings as a sole metric for assessment but believe our research may be used to help create a statistical analysis of the frequency of misunderstandings causing errors. With a larger amount of participants a study could supply examiners with an additional tool to help them gain insight into the understanding of the students they assess.

Acknowledgments

I want to thank my supervisor Ragnhild Kobro Runde for her patient guidance the past year. Thank you for all our interesting meetings where we seem to never stay on topic.

I want to thank Stein Michael Storleer, Stein Gjessing and the rest of the teaching staff of INF1010 for the spring of 2015 for helping me with questions regarding mandatory assignments, expected errors and a whole lot more. I also want to thank Kristin Broch Eliassen for her patience and help giving me quick access to assignment solutions as soon as they were delivered.

I want to thank Kristin Brænden taking the time to read my thesis and give feedback on my writing. I want to thank my family and friends for support on the days when the work with this thesis seemed to have no end. I want to give special thanks to my mother for all the feedback on a lot of the many typographic errors I have produced the last year.

Finally I want to thank everyone at the Department of informatics, especially all the awesome people in the many student associations. Thanks to you most days have been filled with laughs and funny moments. The last five and a half years have been amazing for me.

Contents

List of Figures

List of Tables

Preface

Computers and software have become a vital part of the daily life of most people in developed countries, whether they are aware of it or not. We live in a society where almost every single object we interact with either has or will have a computer or network interface included. The need for experts in the field of computer science shows no sign to stop growing.

The field of computer science may trace its origins all the way back to the abacus invented some time between 2700-2300 BC. But one of the most important pieces of any computer today is the software it runs and that is a far younger invention. The first outline to a computer program was made by Ada Lovelace in the 19th century. But neither it nor the computer it was supposed to run on was ever created. It was not until 1936 when Alan M. Turing published his article, "On computable numbers, with an application to the Entscheidungsproblem," that any theories about software was ever proposed. And still it was first after 1946, and the appearance of the first electronically coded computer, we started to see higher level (non-machine code) programming languages appear.

For this reason the field of Computer Science Education and more specifically Programming Education have only had a few decades to develop. There is a growing number of students in the field of Computer Science. Therefore the field requires much research on how to more effectively teach students how to write program code and engineer large pieces of code.

The last three years I have been a teaching assistant in different programming courses at the University of Oslo. I have had the privilege and responsibility to help students gain proper knowledge to become better programmers. After having seen the effect my own teaching, sometimes good and sometimes bad, I have developed a great interest in how to improve upon it.

This interest lead me to write my master thesis about Computer Science Education. Through my experience with assessing students assignments and exams I learned a lot about what challenges the examiner face. These experiences lead me to want to look at the programming errors made by novice programmers. I have experienced the need for better ways to gain insights into the understanding of a student and saw my thesis as a way to help towards such a goal. My greatest wish for my work would be for it be part of something that makes it easier for new students to learn how to become an expert programmer.

Chapter 1

Introduction

All novice programmers will inevitably make a lot of coding errors as they stumble to become experts. Some of the errors are weeded out immediately as the novices continuously learn how to detect and fix errors in their code. But for programming students some errors persist and will show up in their solutions to assignments and exams. From that point those errors are instrumental in the work of teaching staff and examiners. They are tasked with either helping the students avoid the errors in the future or give the students grades based on their performance.

Unlike an essay or an oral report, program code contains few, if any, direct answers to the authors intentions in writing them. Sometimes the student provide the examiner with comments in the code or a README-file to aid in understanding the program. But that is often limited to explaining the functionality of entire methods, not the intention behind single lines of code. The program code is written in a formal language used to describe a sequence of instructions required to make a computer complete a task. Unfortunately the novice may have misunderstood the syntax or semantics of the language in a multitude of ways and may not even be aware an error exists at all.

In an ideal situation all the feedback a student is given on their work should have the sole goal to help the student improve upon his skill with the topics at hand. That means the person tasked to review the code of the student has to figure out the reason that error is made. It is not always enough to say what parts of a program is erroneous and how it would look like if it was correct. That type of response address the error, but not necessarily the misunderstanding. The response should address the misunderstanding instead and try to correct it. After the misunderstanding is corrected the student should no longer make the same error. But experience shows that due to limited time and skill in reviewing the code of novices the error is addressed and the misunderstanding lives on.

The grades given a student is supposed to reflect his proficiency of the topics in a course. We assess computer programs written by the students to assess the student, not the program itself. What we really wish to assess is the understanding and the mindset that has been used to create the program, not simply whether it produces the correct result or not.

In that regard knowing and understanding the misunderstanding behind the error is key. The error itself is only a symptom of something the student is not suitably proficient at. An examiner needs to know what the misunderstanding is to be able to judge whether the error is relevant to the scoring system or not. Based on this we can assume that identifying the source of errors in program code should have quite an impact on the grade a student is given. Yet examiners do not have a good enough way of doing this today.

Score are therefore often set based on criteria that consider the program instead of the programmer. The last 15 years a lot of research has been made into automatic assessment methods for programming assignments[8, 9, 18, 19, 22, 26, 47, 56]. Many universities now include automatic assessment in their courses to reduce the load on faculty. What we see is an increasing trend where the students are assessed by strict formal demands that the assessment software are given.

1.1 Our goals

The intention of this study is to look for possibilities to to re-humanize assessment of assignments. We want to analyze students solutions to assignments to identify errors and the misunderstandings that cause them. These errors differ from errors that occur while the student is working by not being fixed before they hand in their solutions. The student either believes the program to be complete or have not had the skill or time to improve more upon it. In this thesis our main goal is to answer three questions:

- *Which programming errors do students of a first year programming course make in their solutions to assignments?*
- *Which misunderstandings may have caused these errors?*
- *Is it feasible to use these misunderstandings as an aid in assessing students?*
  - *In formative assessment?* [1]
  - *In summative assessment?*

In this thesis the scope of misunderstandings extend beyond that of misconceptions of certain concepts. It also extends to concepts and principles the students simply do not understand or know of.

Additionally we will consider some aspects of the learning situation that may effect the reasons certain errors are made such as how a course is organized.

[1] Formative and summative assessment are two approaches to assessing students that lead to different outcomes[29]. One aim to identify problem areas to help improve understanding for the students. The other aim to evaluate the students proficiency in the course an provide marks on his performance.

1.2 The scope of the study

We will focus on students learning to program in Java. All the participants in this study are students in a second semester course in object-oriented programming at the Department of Informatics at the University of Oslo in the spring semester of 2015. A further description of the course can be found in section 2.4.

This master thesis was written and conducted as an independent study. Its goals and intentions was developed by the master student and not designed to fit those of the targeted course or the Department of Informatics in general.

The students we take a look at are at a level where preexisting research on their errors is scarce. Their skill level is somewhere between what is typical for students of CS1 courses and students of CS2 courses.[2] The students have become skilled enough to avoid most syntax errors and the simpler types of semantic errors that is common among students of CS1 courses. Yet they are not yet on the level of considering proper algorithmic design in the way CS2 students should be.

1.3 Thesis overview

This thesis is divided into the following parts: In chapter 2 we present some related research in the field of Computer Science Education Research and discuss and customize the terms, measures and definitions used in this thesis. In chapter 3 we discuss our choice of methods, give an overview of what was done to gather data for the study and present the methods used to attain that data. In chapter 4 and 5 we take a look at the errors identified in interviews made with a total of 14 students. Then in chapter 6 we supplement the data found in the interviews through individual observations of nine students solving programming tasks. In chapter 7 we review the errors we identified and take a look at which misunderstandings may have cause those errors. In chapter 8 we attempt to place those misunderstandings into the scope of the course and discuss possible causes and what we may learn from them. In chapter 9 we conclude our findings and take a look at possible future work and improvements of the targeted course.

[2] CS1 and CS2 are terms used by the ACM Computing Curricula to designate the first two courses in the introductory sequence of a computer science major.[30]

Chapter 2

Background

In this chapter we take a look at some existing research and frameworks from the field of Computer Science Education Research as well as terms and constructs used in this study. We will also adapt some of these terms to suit our needs better, creating the base knowledge upon which this study rests. We will also present the university programming course in which this study focus on.

2.1 Related research

2.1.1 Identifying errors

There has been done a lot of research on identifying programming errors made by novice programmers. Several researchers have attempted to identify the most common programming errors students taking CS1-courses make [1, 6, 7, 15, 33, 34]. Many of them focus on how students attempt to fix the errors or how long time they spend on fixing specific errors[1, 15, 35]. Unlike this thesis the research mentioned above tend to focus on the errors that arise in code as the novice is working to complete some task, at the compilation of the code. This differs from the errors that are part of a handed in solution as novices are likely to manage to fix most errors that cause compiler-errors before they hand in an assignment.

2.1.2 Identifying misunderstandings

Some research has been done to identify different misunderstandings that novice programmers have about different topics in introductory level programming courses. Ragonis and Ben-Ari[45] study misunderstandings about object-oriented programming that arise in a course that use the "objects-first" approach to teaching programming. Turner et al.[53] take a look at misunderstandings regarding design principles in object-oriented programming. Danielsiek, Paul, and Vahrenhold[14] and Karpierz and Wolfman[36] attempt to identify algorithmic misunderstandings among students of CS2 courses and link them to specific topics and algorithms and data structures such as binary heaps, hash tables and dynamic

programming. That is not something the novices we take a look at handle yet. Unlike these studies we connect the misunderstandings we identify to specific errors that are common for the students we take a look at.

2.1.3 Assessing programs

A lot of research has been done to improve assessment of programming assignments. We can see a mix of research into ways of manually and automatically assessing programs.

As for manual assessment Ginat and Menashe[27] attempt to use the SOLO-taxonomy[1] to classify algorithmic components and assess the algorithmic design of CS2-students. Fitzgerald et al.[24] take a look at how the examiner's view of his role affects what methods are used to give scoring to a program. Olson[44] shows that different attributes of a student's program code is measured by using either a holistic or an analytic method of scoring computer programs.

On the side of automated assessment a lot of case studies has been made into testing specific automated assessment tools[13, 20, 31]. Carter et al.[9] take a look at the extent of Computer Aided Assessment[2] across different universities and how course planners and teaching staff experience its use and benefits. Wilcox[56] review the benefits and drawbacks of using automated assessment. They experience that student learning may sometimes be impaired by the rigidity of the assessment scheme.

2.2 Defining errors and misunderstandings

In this thesis we focus mainly on errors in program code and the misunderstandings that cause them. Before we go further we need to make a formal definition of the use of the words error and misunderstanding.

2.2.1 Defining an error

The following definition of an error in program code and its relation to other concepts is taken from McCall and Kolling[41]:

- The error is the nature of the problem in the source code that causes the compilation or execution to fail.
- The diagnostic message is the human-readable message presented to the programmer when execution or compilation fails.
- The programmer misconception is the nature of the misunderstanding that resulted in the erroneous code being written.

[1] Learning taxonomies, such as SOLO, are explained in section 2.3.1

[2] Carter et al.[9] defines Computer Aided Assessment as any activity in which computers are involved in the assessment process as more than just an information storage or delivery medium

Our adapted definition

The definition by McCall and Kolling[41] suits our needs quite well. But it does not fit completely. We discovered two types of errors that are important in this thesis but not included in their definition:

- Logic errors in code that do not produce a compilation or execution failure but ends up creating a faulty result.
- Design errors or flaws where either the efficiency, scalability or safety of the program is reduced creating sub-optimal code.

With the new types of errors and flaws in mind the revised definitions to be used in this thesis are as follows:

- The error or flaw is the nature of the problem in the source code that causes the compilation or execution to fail, the program to give an incorrect result or perform in a sub-optimal way.
- The diagnostic message is the human-readable message presented to the programmer when execution or compilation fails.
- The programmer misconception is the nature of the misunderstanding that resulted in the erroneous or flawed code being written.

As we can see here the diagnostic message only covers some of the errors and flaws a programmer can may have in his code. This is the most important reason why we will not focus on the diagnostic message at all in this thesis.

The definition of a programmer misconception helps us see the relation between misconceptions and misunderstandings. Misconceptions only cover erroneous understandings of topics or concepts. In this thesis we will focus on the broader perspective that misunderstandings provide such as the lack of any understanding at all.

2.2.2 Categorization of errors and flaws

In addition to an exact definition of what an error or a flaw is we need proper definitions of different classes of errors and flaws.

Categories from existing research

In an experiment to identify and help novices programmers understand some of the most common errors made when writing Java code, Hristova et al.[33] categorizes errors into syntax errors, semantic errors and logic errors. They consider the first to be caused by spelling mistakes, bad punctuation or order of words in the program. Semantic errors refers to errors that deal with meaning of the code such as mistaking how certain instructions works. Logic errors are the ones that arise from faulty thinking by the programmer. These may also manifest as bad syntax or semantics.

So the border between them is not completely clear cut. Hristova et al.[33] encountered the problem as some errors seemed to belong to sometimes two and even all three categories. These categories have also been used by other researchers[1, 6].

In their work to identify the most common syntax errors and time spent solving them both Denny, Luxton-Reilly, and Tempero[15] and Jackson, Cobb, and Carver[34] choose to only use a narrower classification usually linked to either the diagnostic message from the compiler or the "reason" the compilation failed. Examples of categorizations are: *Cannot resolve identifier*, *Missing ;* and *Using.length as a field*.

Our adapted categories

The set of categories that fit our work best was the one made by Hristova et al.[33]. It provides us with broader categories that can be applied to our needs quite well. In their work Hristova et al. say they made the classifications as an attempt to separate errors based on the thought process of novice Java programmers. Others have used those definitions to analyze programs. But those studies focus on simpler errors made while writing and compiling code. The code we analyze is from solutions to assignments. Our method involves classification of errors in code with consideration to how they are detected and the impact they have on the Java program. To fit those needs we made some adjustments to the categories of Hristova et al.

Modified to match our needs the definitions are more explicit and make a clearer distinction between the three categories that cause a program to not compile or give an incorrect result. We also include the definition of design errors and design flaws included in the definitions of errors and flaws in section 2.2.1.

It is important to understand that the line that separates design errors and design flaws is highly context sensitive. Different courses and teaching staff can place that line at different places.

- Syntax errors: spelling mistakes, punctuation and the order of words in the program causing compiler failure.
- Semantic errors: non-syntactic errors ending in either compiler failure or failure of the runtime execution.
- Logic errors: errors that result in an incorrect result at runtime.
- Design errors: errors that arise from bad choices considering program efficiency, scalability and robustness, that students of the course are expected to know based on the learning objectives of the course.
- Design flaws: errors that arise from bad choices considering program efficiency, scalability and robustness, that are outside the scope of the course but more experienced programmers would avoid.

Classifying errors within the first three categories is now quite easy. The hard part is to decide whether an issue with design is an error or a flaw. It

requires good insight in the targeted course and how it is taught. In this study we had the help of some of the teaching staff of the targeted course to aid us in reviewing the classifications we use in chapter 4, 5, 6 and 7.

2.3 Elements of a university CS course

To understand the importance of errors and understandings in a CS course we need to know which elements a course is based on. The elements given below are in themselves not tied directly to any specific principles for course design. But the interaction between them follows the principle of Constructive Alignment[2].

2.3.1 Learning objectives

The very first thing a student should be presented with when starting a new course should be the course learning objectives. A learning objective is a short description of what the students should know or be able to do at the end of the course that they were unable to do before. The learning objective should be covering both what topic it covers (e.g stacks or binary search trees) and to which extent the student should be familiar with the topic (e.g be able to use a stack in different algorithms, or understand how binary search trees improve search speeds in comparison to list structures).

Learning objectives should be used by course designers to help deciding which topics should be included in the teaching activities of the course, what should be included in the curriculum and also which teaching activities are appropriate.

Learning taxonomies

To be able to formulate good learning objects course designers need to be able to classify and formulate the depth of understanding required for an action.

Learning something new is all about understanding a topic and the concepts it is based upon. But the word is somewhat problematic when used to describe wanted outcomes from a university course. 'Understanding' may have different meaning to different people and in different contexts. Entwistle, Entwistle, et al.[21] show that understanding can be separated into different levels based on the subjects' ability to structure their answers about a topic.

Over the years a lot of different learning taxonomies have been proposed. The most popular ones today are possibly Bloom's Taxonomy[5] and the Structure of the Observed Learning Outcomes (SOLO)[3]. They share similarities by being hierarchically built as the higher levels encompass all understanding of a lower levels. But they differ in both number of levels and how the levels are designed. Both taxonomies are highly dependent upon the learning context when it comes to classifying a certain action or task based on the understanding required. Something that merely

requires recollection from someone who has seen that very example before may require more analytical thought from someone who knows only the concept used to complete the task.

Learning taxonomies are tightly bound to classifying learning objectives and assessment. Yet they are often difficult to apply and require the learning objectives to be well defined and understood. This difficulty creates possible discrepancies between how topics are taught and how they are assessed.

While these general taxonomies have been used in the design of courses for decades, they have not been adopted as wholeheartedly by Computer Science educators. A lot of research has been done in attempting to use the taxonomies in both writing learning objectives and assessing students and their understanding [10, 28, 38, 48, 50, 51]. But some researchers in the field of Computer Science Education feel these are inadequate to describe learning outcomes in Computer Science. Fuller et al.[25] attempts to change the taxonomies to adapt them to Computer Science by splitting Bloom's Taxonomy into two dimensions, Producing and Interpreting but are not completely happy with the result.

2.3.2 Teaching activities and learning activities

It is common to separate between the concepts of learning activities and teaching activities. Teaching activities are the activities the teachers do to communicate knowledge to the students, such as lectures, lab sessions and even private conversations. Learning activities are the activities done by students in which they acquire knowledge and understanding about the topic at hand. Biggs and Tang[4] states that teaching activities should be chosen to stimulate the right kind of learning activities for as many students as possible.

The learning objectives should be used to decide which learning activities are relevant for the students to engage in to acquire proper understanding of a topic. Based on this a decision can be made on which teaching activities are best suited to engage the student in those learning activities.

2.3.3 Assessment

The learning objectives should guide exam writers and others who create assessments schemes - from now on denoted instruments of assessment - for the course in how to assess the proficiency of the students. The instruments of assessment should be able to give a reflection of the proficiency of students compared to the intended learning objectives. This should also in the end be reflected in how grades are given by examiners.

The intended learning objectives are the learning objectives presented to the students. The actual learning objectives are the learning objectives used to assess the qualities of the students. These are usually not written and presented to students. They are the result of how writers of

instruments of assessment and the staff grading the students choose to prioritize different elements of the curriculum.

Because of this most courses will have a certain discrepancy between the intended and the actual learning objectives. This may be caused by difficulty in assessing the complexity and difficulty of different parts of an instrument of assessment. It may also be that the intended learning objectives are too poorly written, not up to date or that exam writers do not consider them when grading.

Summative assessment vs. formative assessment

There are mainly two different ways to assess students, formative and summative assessment.[29]

- *Formative assessment* is used for two reasons. To help students identify their strengths and weaknesses and target areas that need more work. To help teaching staff recognize where students are struggling and address those problems.
- *Summative assessment* is used to evaluate student learning by comparing it against some standard or benchmark and give marks on how proficient they are.

The reasons for using these different methods should be considered by course planners when they design which instruments of assessment are to be used in the course. Research into the use of different types of instruments of assessment show that an instrument should do either or the other and not attempt to do both. Lauvås[37] explains how mixing them cause the instruments to achieve less towards at least one of the goals, student learning or grading properly. But in practice we see a lot of courses having hybrids between the two. Whether this is due to time limitations or poor course design is hard to say.

The reason the two should be used separately is the fact that students prepare and work different when the goal is to get as good grades as possible. Students tend to prepare for summative assessment using so called surface approaches to learning, focusing on remembering facts and details in anticipation of the questions they will be asked. Using such methods may cause a lesser level of understanding than using the deep approaches to learning which focus on understanding a topic in depth.

The examiner and methods of assessing programs

Fitzgerald et al.[24] show that different examiners have different approaches towards grading summative instruments of assessment depending on what they see their purpose as examiners to be. Some see themselves as gatekeepers tasked with ensuring only students with satisfactory proficiency advance to later courses. Others see their task as making sure the correct score is given based on the instrument of assessment. Their research show that this can impact how they grade students with solutions

in the middle range. The best and poorest students are shown to be treated very similarly not matter the approach used.

The examiner usually fall to one of two methods to analyze an answer to an instrument of assessment. The examiners using a holistic method grade the students based on their overall impression of the student's answer. The examiners using an analytic method bases the score on an analysis of the constituent parts of the answer assigning them different priorities and weights.

Formative assessment faces the same issue to a lesser extent. As the goal of formative assessment is not to give some metric of student proficiency the focus of the examiner is how to give proper feedback. But due to large classes and limited time to review code the examiner is required to prioritize different aspects of a program as well.

2.4 The targeted course

The subjects for this study are students from a second semester course in object oriented programming, called INF1010, held by the Department of Informatics at the University of Oslo. The programming language used in the course is Java. From now on INF1010 is denoted 'the targeted course'.

2.4.1 Intended learning objectives

The following points are the written learning objectives stated on the course web page[16]:

After completing this course you will have a thorough knowledge about, and be able to use yourself when you program:

- subclasses, abstract classes, interfaces, virtual methods, abstract data types and alternative implementations
- cooperation between objects, including programming with server-clients and peer-to-peer programming.
- some important data structures such as one way and doubly linked lists and binary trees with associated algorithms

After completing this course you will have a good knowledge about, and be able to use yourself when you program, simpler versions of:

- recursion
- graphical user interfaces with event programming
- parallel programming, shared data, synchronization and threads
- the Java class library

2.4.2 Course prerequisites

The students attending the targeted course have some prior experience with programming. Either through the course INF1000, which includes Java programming and a simple introduction to some object oriented principles, or through the course INF1100, which is a mathematics oriented course using Python as programming language.

Because of the different backgrounds, some students have less experience with Java than others. But the assignments we have reviewed were solved over halfway through the semester. The course is set up to allow the Python students to catch up with the others. But they do have a slightly more difficult start to the course. So whether all the students do catch up is hard to know.

This may be a factor to why some students have certain misunderstandings but is outside the scope of this study as we do not compare the background of the subjects.

2.4.3 Teaching activities and assessment

The teaching activities in the course include weekly lectures with the professors, plenary programming sessions and seminars in smaller groups (up to 40 students) with the teaching assistants. None of these activities are compulsory.

The course includes 11 mandatory programming assignments throughout the semester. All students are graded with a numerical score on each task. To be allowed to attend the final exam a student must achieve a scoring of minimum two thirds of the total possible score for all 11 assignments. The final grade is decided by the final exam only, not the mandatory assignments.

2.5 Object-oriented programming

As object-oriented programming (OOP) is an essential part of the targeted course we take a quick look at the background and concepts that make up the paradigm.

Object-oriented programming is a programming paradigm based on the concept of "objects". Objects are data structures containing fields of data and methods defining the behavior of the object.

A programmer using OOP utilize four concepts to design and write object oriented programs. These are the simple, yet very powerful, concepts encapsulation, composition, inheritance, and delegation of data between the objects.

- *Encapsulation* of functionality and data within a class protecting the data and implementation from the rest of the "world".
- *Composition* of objects as some objects contain other objects in their instance variables.

- Creating specialized classes through *inheritance* of attributes from an encompassing concept.
- *Delegation* of data and functionality as one object relies on another to provide a certain set of functionalities.

These concepts make up the core of object-oriented programming and design independently of which programming language is used.

2.5.1 Approaches to OOP

Object oriented programming started out with the creation of the first object-oriented programming language, SIMULA[43], and has since the 1970s been the dominant paradigm used in software development. But the paradigm has developed with two different views on what is important when we are designing a program. The views are not always mutually exclusive and share many traits yet programs may differ a bit in design when considering the two different goals.

The separation between the concepts and the approaches may be a bit unclear when we consider them in object-oriented design (OOD). In this thesis we talk both about applying the concepts but also why they are applied. The targeted course do mainly base itself upon the Scandinavian approach, but it does not express this explicitly. Therefore students' motivation behind their design may follow the Scandinavian approach.

The Scandinavian approach

The Scandinavian approach[43] is an many ways flavored by the way SIMULA developed. The first version,SIMULA I, was created to be a simulation language. But many of the concepts used in SIMULA I was refined in SIMULA 67 that was a full fledged object-oriented programming language.

The approach that was used in developing SIMULA 67 and has become the standard Scandinavian approach to OOP focus on mirroring the part of the world that is relevant to your program. This includes how the constituent parts can be divided into concepts with specific actions and attributes in common and how instances of the concepts communicate to make up the world as a whole. A program designed using the Scandinavian approach tend to follow rather strict rules of how classes are designed and what traits need to be inherited and which can be rewritten by the specialized classes.

The American approach

The American approach[40] started out with the concepts developed with SIMULA but has a different goal than the Scandinavian approach. The goal is more focused on the practical aspect of reducing the amount of code written. This make the limitations to inheritance fewer. Programs written using this approach tend to be more open to contain alternatives to

the "natural" concepts that the Scandinavian approach prefers. Alternative structures are created with the programmer and his work in focus rather than the world.

Chapter 3

Method and methodology

In this chapter we discuss the choice of methods used in the study. We also describe those methods and how they may affect the validity of our results. We also take a look at the time schedule of the study and how we dealt with some of the practical requirements of running a study where you gather personal information.

3.1 Possible sources of data

There are multiple ways to gain insights about novice programmers and the errors their code contain. Common for them all is that we somehow need to assess their assignments.

Looking at our possible sources for data we have several to choose from. We can use code written in students' solutions to assignments, the error messages they produce or other information from the students who wrote them.

Looking at the error messages we do not find them suitable for our needs. Not all errors that are covered by our definition of errors and flaws will output an error message. In addition McCall and Kolling[41] concludes that a single error may, in different context, produce different messages and the same diagnostic message may be produced by entirely different and distinct errors. This all depends on the compiler and run-time environment. The diagnostic messages do not get us any closer to gaining any insights into the reasons why a student ends up with a certain error in his program. It may be a reason for the error in itself as some students may have issues with bad diagnostic messages, but they do not help us identify misunderstandings. For that reason the diagnostic messages are a poor source of data for our study.

Our main goal for the study is made up of three parts. The first is to learn more about which errors students make in their assignments. The second is to learn about the reasons why they make those errors. The third is to attempt to answer whether it would be feasible to use misunderstandings as an assessment metric in the two types of assessment. The first of these is quantitative of nature, but the second is qualitative. The second goal is by far the most important in this study. The

misunderstandings that are identified would answer our second research question and an analysis of them can possibly answer the last question. For that reason the study was designed with more weight on the qualitative data collection than the quantitative. This makes it a qualitative, not mixed method, study[12].

Fincher and Petre[23] states that the granularity of your data sources must match the level of resolution of the data you need. We need data from individuals so we can study their method of solving a programming task. We have quite a few rich data sources available, such as student solutions to assignments as well as the students themselves.

3.2 Designing a qualitative study

According to Creswell[12] the traits of methods for collecting qualitative data is open ended in nature. That means it cannot be captured through any standardized questionnaire or other non-flexible extraction method. The data may take an unexpected turn and requires a certain amount of flexibility of the method used. The questions asked or setting the subject is put into need will usually adapt as new knowledge is gained about the subjects' behavior.

That reduces the different methods available to us. According to Creswell[12] data collection procedures adapted for qualitative research involve at least one of four basic types. Creswell describes these in the following way[12, p. 185]:

1. *Observations*, in which the researcher takes fieldnotes on the behavior and activities of individuals at the research site. On these fieldnotes, the researcher records, in an unstructured or semistructured (using some prior questions that the inquirer wants to know) way, activities at the research site. The qualitative observer may also engage in roles varying from a non-participant to a complete participant.

2. In *interviews*, the researcher conducts face-to-face interviews with participants, interviews participants by telephone, or engages in focus group interviews with six to eight interviews in each group. These interviews involve unstructured and generally open-ended questions that are few in number and intended to elicit views and opinions from the participants.

3. During the process of research, the qualitative investigator may collect *documents*. These may be public documents (e.g., newspapers, minutes of meetings, official reports) or private documents (e.g., personal journals and diaries, letters, e-mails).

4. A final category of qualitative data consists of *audio and visual material*. This data may take the form of photographs, art objects, videotapes, or any forms of sound.

Some of these definitions may be a bit strict and out of date. E.g. interviews may involve semi-structured questions instead of unstructured ones. New technology and use of the Internet has also introduced a lot of new communication tools which can be used for observations and interviews.

From the basic building blocks described above we devised three possible strategies that could be used. The strategies are described below and pros and cons are given in Table 3.1 on the following page. Some of them are general, taken from Creswell[12] and Fincher and Petre[23], others consider issues specifically related to our study.

3.2.1 Work-logs

With this strategy the subjects are tasked to log their work process as they work on an assignment we want to review. The logs serve as written reports that describe the subject's thought process as he completes different programming tasks. It would list the subject's progress, issues and why he believes those issues appear. Together with the subject's solution to the assignment the work-log provides the entire data set.

3.2.2 Interviews

With this strategy the researchers review solutions to programming assignments to identify errors and interesting parts of the solution. The subjects with interesting errors in their solutions to the assignment are invited to an interview where they explain their process of solving that assignment. The interviewer allows the subject to control the conversation but ensures the interesting areas of the solution is explained. The conversation covers all interesting parts of the subject's solution to the assignment and his reasons for choosing solution method. The interview audio is recorded to allow the researcher to revisit the interview in detail. Together with the subject's solution to the assignment and the interviewers notes, the audio provides the entire data set.

3.2.3 Observations

With this strategy the subjects participate in individual observations of them as they solve either parts of, or an entire programming assignment. The observation reveals the choices made by the subject step by step. The observer can either be passive or active, by choosing whether to ask questions or not, but does not participate in solving the assignment. The observation audio is recorded to allow the researcher to revisit the observation in detail. Together with the subject's solution to the assignment and the observer's notes, the audio provides the entire data set.

	Pros	Cons
Work-logs	• Enables us to obtain the language and words of subjects • Can be accessed at a time convenient to the us • Represents data that are thoughtful, in that subjects have given attention to writing down	• Requires a lot of attention from the subjects. In our case over a two to three week period. • Requires proper instruction from the subjects in how to write work-logs and which thoughts and choices we are interested in. • Does not allow for selection of subjects before the work-log is collected • Time consuming analysis due to the richness of data
Interviews	• Enables us to obtain the language and words of subjects • The subjects may disclose thoughts and actions happening over a longer time period than the interview lasts • Allows us to control over the line of questioning • Allows for selection of subjects before the interview is conducted	• Provides us with indirect information filtered through the views of the subjects • Removes the subject from the natural setting in which the actions are conducted • Our presence and questions may bias responses • Not all subjects are equally articulate and perceptive • The information is recorded quite a bit after the programming is conducted and may be effected by the subject's ability to recall his actions • Time consuming analysis due to the richness of data
Observations	• We can record information as it is revealed • Unusual aspects of the solution can be noticed during the observation • Subjects may disclose uncertainties and lacks of knowledge they otherwise would not be comfortable to discuss	• The subject may find the observer intrusive and not behave in a normal manner • Requires skills at attending and observing from the observer • All actions and thoughts that are to be recorded must happen during the observation • Does not allow for selection of subjects before the observation is conducted • Time consuming analysis due to the richness of data

Table 3.1: Pros and cons of proposed data collection procedures

3.3 Methods used in this study

In the introduction we stated that one of our goals is to learn which errors students make in their solutions to assignments or exams. Choosing mandatory assignments as the instruments of assessment that we are to review gives us a huge benefit. The mandatory assignments are solved in a setting equal to the one we wish to analyze. There is no need to pay special attention into designing a task or experiment to ensure the students work in the way we wish to capture.

The students are given between two and three weeks to solve each assignment. That makes them fairly long and time consuming. With that in mind we had a good basis to make a choice between the different strategies we had proposed. We used the list of pros and cons given in Table 3.1 on the preceding page together with the practical limitation given by our goal to choose a strategy.

Since we do not look at shorter assignments such as classroom exams our data will only be applicable to longer assignments such as the ones we review.

Considering that the errors we were looking for are errors present in solutions to assignments the choice of subjects became important. We did not expect all students to have enough if any of these errors. This makes the opportunity to weed out the students with no interesting errors a valuable trait of the interview strategy.

We wish to explain one particular weakness with using observations mentioned in Table 3.1 a bit further. The fact that the subject may find the observer intrusive is something that may lead to a huge bias from the very nature of someone knowingly being observed. The bias comes from a reactivity called the Hawthorne effect, sometimes also called the observer effect. It is a type of reactivity in which individuals modify or improve an aspect of their behavior in response to their awareness of being observed[39]. For example this may cause the participants to attack a problem from more angles before choosing a solution than they normally would.

The size of the assignments also made observations of the assignments as a whole unfeasible. It would take too much time, both for us and the students. It would also be unlikely that either observer or student could remain alert for a long enough time period. Work-logs could provide good data. But we were not confident in being able to train the students properly in writing them. It may also have affected our ability to recruit participants as it requires a bit more work than an observation or interview.

3.3.1 Interviews about solutions to assignments

Considering the nature of the assignments interviews proved most promising for our study. Many of its advantages could be exploited in the data collection. We are able to choose our subjects based on criteria that mark them as interesting for our study before we commit to interviewing them. It enables us to obtain the language and words of subjects, giving us in-

sight into how they think about problem solving when writing computer programs. The subjects may disclose thoughts and actions happening over a longer time period than the interview lasts. This allows us to possibly gain access to all they have done to solve an assignment in the targeted course. It also allows us to control over the line of questioning to minimize the amount of work in analyzing the results and avoiding topics that are not interesting.

The most important cons, such as possible bias from the interviewers' questions, can be reduced by carefully planning the interview. The cons of not getting data in real time can be reduced as much as possible by ensuring the interviews are conducted as soon as possible after the assignment is handed in. We cannot, however, ensure that the subjects will always speak truthfully.

The methods used in the interviews will be explained in section 3.4. Details of how the interviews were conducted are provided in chapter 4 and chapter 5.

3.3.2 Supplemental data collection using observations

To supplement the results from the interviews with data without the loss of detail that may arise from interviews being held after the assignment was handed in we decided to conduct some observations as well. The data from the interviews requires us to guess at their misunderstandings based on statements given as much as a couple of weeks after they worked on a particular problem in an assignment.

The observations allows a more detailed view of how they work and may support our belief on certain misunderstandings students may have. It is important to remember the biases caused by the Hawthorne Effect mentioned in section 3.3. For that reason we cannot be sure if the students do not change their method of problem solving as they are being observed. So we will avoid making conclusions that discard certain misunderstandings or issues the students show signs of in the interviews.

We designed a shorter set of tasks for the observations, solvable within a time frame that is suited for the format. The methods used in the observations will be explained in section 3.5. Details of how the observations were conducted are provided in chapter 6.

3.4 Interviews

The interviews make up the main bulk of our data collection. They are focused on errors made in assignments in the targeted course. This helps us reduce the possible biases considering the main goal with this study. We wish to identify errors and misunderstandings in the solutions students submit to instruments of assessments such as programming assignments and exams. Mandatory assignments fits that description so we know the students work in the way that is natural for them in that kind of situation.

An added benefit is the targeted course requires students to complete them. This makes recruiting subjects for the study easier. No additional work had to be done by the subjects unless they were invited to participate in an interview.

3.4.1 The choice of assignments

We wanted to review two different assignments giving us an opportunity to test our methods before we committed to it. For that reason the interviews are done in two rounds. The first round consists of only four interviews. Yet the data we acquired in that round is usable in our results. The subjects are the same in both rounds of interviews, though some were recruited to the study after the initial interviews were conducted. So the data from both rounds are applicable in the study. The assignments that are reviewed include different topics so the two rounds will mostly result in identifying different errors. But the misunderstandings causing those errors may overlap.

The assignments are both "take-home" assignments spanning two to three weeks. They are mandatory and students need a certain amount of them approved to be allowed to attend the final exam. They are not part of the grade the students are given. This may have an effect of how students work to complete these assignments. Our results may be affected by this and the errors and misunderstandings we identify are not necessarily the same as in a four our written classroom exam. In shorter exams the students are likely to make more serious mistakes and not have time to fix them. Similarly the longer assignments may include more difficult tasks than a short exam that challenge the students on different skills.

As the study required some preparations (see section 3.9.2 and 3.10) we chose from assignments planned for the latter half of the course. It was important that the topics being tested included topics that are core in this course and likely to be part of most introductory programming courses. That removed an assignment being solely about graphical user interfaces.

We wanted to optimize our chances of getting as many participants as possible. Therefore we discarded using assignments due just before holidays and too close to the final exam. The timing would have made it harder to have the students participate in interviews. Also considering the time needed to analyze and process the results between the two rounds of interviews we made a good choice of assignments. The assignments themselves are summarized in section 4.2 and 5.3. The assignment texts can be found in Appendix E and F

Assessment method of the assignments

From talking to the professors responsible for planning the targeted course we were told the intention with the assignments is to make the students work with the topics in the curriculum throughout the entire semester and give feedback to what their problems are. That would make the assignments instruments of formative assessment.

But before we take that for granted we need to consider the scoring system and the points requirement described in section 2.4.3. A scoring system is a trait often seen in summative assessment. The fact that students can fail the course based on their scores make it possible to argue that the assignments are part of the material that is used in their final grade. Failing to get the required score will technically give them a failing grade, even if they do not get to take the final exam. From that we concluded that assignments are hybrids between formative and summative assessment.

That may have an effect on how students work with the assignment. According to Lauvås[37] students may cope with continuous summative assessment using lesser learning activities such as copying and pasting code from the Internet and blind trial and error to achieve the output they want. This may cause long term learning to be reduced.

3.4.2 Preparations

Before we stated identifying errors in the students' assignments we analyzed the assignment given to them. As the mandatory assignments in the targeted course are rather large, we chose to review the students' solutions first and identify errors in their code as well as classifying them according to the categories we created in section 2.2.2. The students whose assignments contain errors that are interesting were invited for an interview.

3.4.3 Using retrospection

The method chosen for the interviews is called retrospection[54]. When using retrospection the subject for the interview is questioned about the thought processes during the solving of a problem he has solved. For example he may be asked: "How did you ensure the linked list do not contain any duplicates?"" The questions may also be more detailed, like "Which data structure did you use?". This method was chosen over methods such as prompting and questioning to ensure the subject was asked as few questions as possible. This consideration was made as the questions themselves may be leading and several key points may be lost due to the way the questions are formulated.

Retrospection has some weaknesses. The subject may not always remember exactly what he did, especially if some time has passed after completing the task. The subject may not even be aware of what he is doing as he may not have a conscious process for solving the problem. The subject may also present his thought process as more coherent and intelligent than it originally was. This may happen both intentionally and unintentionally as the subject may not remember his original process. His memory may by guided if his knowledge of the problem has changed since the he solved it. This may cause some misunderstandings to be lost. But any positive proof of misunderstandings should not be impacted by this.

At the time the interviews were planned it was not certain how many from the research group would interview the students. A detailed

interview guide was written to ensure all interviewers followed the same procedure. It can be found in Appendix B[1]. In the end all the interviews were performed by the author of this thesis, but the interview guide was still used in all the interviews to ensure the author followed the same procedure in all the interviews.

3.4.4 Initial interviews

Trying to ensure our interview methods would give the best possible results, the interviews were split into two rounds. The first round was intended as a trial run with a smaller amount of participants. The second round included more participants and revolved around a different mandatory assignment than the first. Yet we still managed to get good results from the initial interviews.

3.4.5 Tools and location

To complete the interviews we used an audio recorder and post-processed all the interviews using the audio playing software VLC.

All interviews were conducted at the university with only the interviewer and the participant present.

3.5 Observations

To supplement the data from the interviews our goal with observing students work became to learn as much as possible of how the students were attempting to solve a problem. More specifically we really wanted to get a feeling of the level of understanding they had of the different topics involved in the tasks.

For that reason we could not rely on the visual aspect of watching them write code. The reasons behind their choices may still be uncertain even if you know their choices and actions. Therefore we had to involve the student in a different manner. To get insights into their thoughts we wanted them to speak of their actions and choices while they were making them.

3.5.1 The think aloud method

The observations were conducted according to the think aloud method[54]. This method requires the subject to continually speak out loud everything they are thinking as they are working. The observers job is to make sure the subject does not stop verbalizing their thoughts using questions such as "What are you thinking now?", "What are you doing right now?", "Why did you do that just now?". The questions are there just to keep the subject talking throughout the observation.

[1]The interview guide in Appendix B is the one used for the initial interviews. The errors we were to look for were changed for the second round of interviews

The think aloud method was chosen over other observation methods such as retrospection, introspection and questions and prompting because of its strengths in letting the data gained remain unbiased by the observer's questions and at the same time not lose any details from the subject forgetting what he though about.

Pros

- The think aloud methods does not require waiting until the subject has completed at least a part of the task like retrospection and introspection do.
- The think aloud methods does not require the observer to choose when the right time is to have the subject explain their actions like introspection and questions and prompting do.
- The think aloud method keeps the questions asked by the observer to a minimum avoiding leading questions that create biased data or questions that lead to false answers as the subject may answer what he believes the observer wishes to hear.
- The think aloud method avoids the subject unconsciously cover up their mistakes as soon as he has figured out what he had though wrong as the human brain has a tendency to also forget what we though wrong as soon as we figure out what was truly correct, creating false memories[54, p. 44].

There are some weaknesses to the think aloud method mainly regarding the importance of choosing good tasks and having subjects that are good at verbalizing their thoughts.

Cons

- The think aloud method requires the subject to be good at verbalizing their thoughts and actions. Those who are bad at it simply do not fit the method and leaves you with little data gathered.
- The think aloud method requires the problem solved by the subject to not be too easy or too hard. Too easy tasks are solved on an unconscious level and such thoughts are impossible to verbalize. Too hard tasks cannot be solved at all and will yield few results.

3.5.2 Tools and location

The observation audio was recorded using an audio recording device. The observer was taking notes and the recording was later reviewed by the observer using the notes as backup.

All observations were conducted at a meeting room at the university with only the observer and the participant present. The students were allowed to write the code on a computer of their choosing. For most

students this meant the subject's own personal laptops. This was to make the problem solving as authentic as possible, having all the same resources available as when they are solving an assignment in the targeted course such as code they themselves has written. Due to difficulties in distributing the software we chose to not have any software recording the screen during the observation.

3.6 Data analysis

To analyze the data from interviews and observations we use a qualitative approach[23].

The analysis is made in two passes through the data. First we analyze the data from each individual attempting to isolate the individual misunderstandings. We use the errors identified in the subject's assignments as a guide to which parts of the recorded data are interesting. The interesting parts are then more thoroughly searched for patterns that suggest lack of or erroneous understanding that may explain the errors that was experienced. At the same time we search for patterns that may disprove the suggested misunderstandings. If a subject's statements indicate that the he seem to understand the topic properly the misunderstanding may need to be discarded.

After identifying what we believe to be likely individual misunderstandings we cross-reference the misunderstandings that are identified in the different interviews. In cases where a certain error or misunderstanding appears for multiple subjects, we compare the data and use clear similarities to strengthen the likelihood that the misunderstandings may be generalized for novices of the targeted course in general. The skewed selection of students do make this generalization limited to the students who have more errors in their assignments.

The errors and misunderstandings that yield the most data are selected as most likely to be common among students and presented in detail in chapter 7.

3.7 Measurements for statistical analysis

This thesis mostly focus on qualitative data. So it does not contain much use of statistics. But in preparation to the observations we use a simple statistical analysis to help us choose appropriate exercises. The measurements used are the basic statistical measurements mean, mode, median and standard deviation. Their formal definitions are defined in table 3.2 on the following page.

3.8 The subjects

A total of 41 students in the targeted course participated in this study. Of those 41 students 23 participated in either an interview or observation. One

Name	Denoted	Explanation	Formula
mean	$\bar{x}$	The average of the data set	$\frac{x_1+x_2+...+x_{n-1}+x_n}{n}$
mode	mode	The most frequent value in the data set of all gathered data	x_i when $\forall x(count(x_i) \geq count(x))$
median	$\tilde{x}$	The middle value separating the lower and upper halves of the data set	$x_{(n+1)/2}$ when $\forall x(x_i \leq x_{i+1})$ and $n\%2 = 1$
			$\frac{x_{(n/2)}+x_{(n/2)+1}}{2}$ when $\forall x(x_i \leq x_{i+1})$ and $n\%2 = 0$
standard deviation	s	The average difference from the mean in the data set	$\sqrt{\frac{(x_1-\bar{x})^2+...+(x_n-\bar{x})^2}{n}}$

Table 3.2: Statistical measurements used in this study

of the students participated in both rounds of interviews.

The students were recruited via email asking them to participate in the study. This meant giving the research group access to their answers to mandatory assignments in the targeted course, as well as their final grades upon completing the course.

Select students were also invited to participate in interviews or observations as explained in section 3.4 and 3.5. As our focus was to identify errors in answers to instruments of assessment, that selection was a bit skewed considering skill level. The higher level students don't get that many errors in their solutions. Therefore we invited mostly lower- or mid-range level students. This should not have a huge impact on our results as we are exploring the misunderstandings behind certain errors. Those who don't make the errors would not add any data to that specific case. They may share the misunderstandings of the students who make the errors and also have completely different misunderstandings. But those misunderstandings cannot be connected to specific errors and are therefore difficult to use when considering them as a metric to assessment. They would be very interesting to use to test how they might impact a scoring system based on misunderstandings, should it ever be created.

3.9 Handling subject privacy rights

In this study we wanted to review the solutions to assignments in the targeted course. The intention was to invite the students who made them to interviews or observations. For that reason we needed to store some personal data such as who the authors of these solutions are and how to contact them.

As we stored information that can identify students in the course and their solutions we needed to have systems to handle the privacy rights of those who chose to participate. This includes making the direct data sources anonymous and replacing any personal identifiers such as usernames or email addresses with unique numerical keys. A reference list containing the numerical keys and the identities of the participants was then stored separately from the data.

All subjects had to approve being part of the study before we were given access to their solutions on the specified assignment and contact information to be able to invite them to interviews and observations. They were all given a Participant Information Sheet. The sheet discloses all the specifics about study and what the data will be used for as well as privacy handling. These sheets were signed by all the participants upon signing up for the study. The sheet can be found in Appendix A.

The study was reported to and approved by the Data Protection Official for Research[2] with the project number: 42265. Their letter of approval can be found in Appendix C.

3.9.1 Independence from the targeted course

The study was conducted separately from the targeted course. That meant that no teaching personnel or examiners were given any data that could identify any of the subjects. But we have consulted with the professors and teaching assistants to do a better job at analyzing assignments and their intentions.

The author who was responsible for the interviews and observations was a teaching assistant in the targeted course during the time the study was conducted and had attended the targeted course before. Certain steps were taken to remedy any biases that may have arisen from having previous knowledge of how the students perform and remove any conflict between tho roles of teaching assistant and researcher. That step was omitting from the study all students whom he had responsibility for either teaching or grading.

This disqualification of some students should not have any effect on the selection available for the study. The targeted course consisted of 14 different groups of approximately 40 students that were not assigned based on skill or grades. Disqualifying one group leaves approximately 560 of 600 students.

3.9.2 Formal approval of the study

Before we could start recruiting subjects for the study or get a hold of answers to instruments of assessment to review we needed approval of our methods to ensure privacy. The instances whom we needed formal approval from were:

[2]Personvernombudet for forskning ved Norsk samfunnsvitenskapelig datatjeneste (NSD)

- The teaching staff of the targeted course
- The administration at the Department of Informatics
- The data administrators at the University of Oslo
- The Data Protection Official for Research[3]

Getting approval from the local instances did not require too much time and was done with a couple of emails in a matter of two weeks. But the Data Protection Official for Research required 30 days to process the application. Because of this a six week period went into writing the application, getting approval from local instances as well as waiting for the application to Data Protection Official for Research to be processed. This pushed the entire start of the data collection of the study up to the middle of March.

The approval letter from the Data Protection Official for Research can be found in Appendix C.

3.10 Overview of the timeline

This master thesis was started in the beginning of January 2015. The thesis spanned over two semesters giving a total of ten months to finish. The final result was handed in in the beginning of November 2015.

The targeted course started in the middle of January and lasted until the summer holiday in the middle of June. That placed a lot of our work with collecting data quite early in the year.

Our first task was to use existing research in the field on Computer Science Education Research to identify goals for the study. Following that the study in itself was designed.

The study required formal approval from multiple instances, some requiring up to a month to process the application for approval. That time was spent preparing the initial interviews. The assignment text was analyzed. Expected errors were identified and a time frame for the next months was set. As soon as formal approval was granted we started recruiting participants for the study. Shortly after the initial interviews were conducted and analyzed.

After a short time analyzing the results and reviewing our method for the initial interviews the next round of interviews were conducted. During the summer our results from the interviews were analyzed and the plan for the observations was finished. Participants for the observations were recruited and in August the observations were conducted as the final procedure for data collection. Our approach at reviewing and updating our goals and methods used in data collection can be found in Figure 3.1 on page 32. The first node from the top in the flow chart represents the work described in this chapter. The second and third nodes describe the process

[3]Personvernombudet for forskning ved Norsk samfunnsvitenskapelig datatjeneste (NSD)

of interviews disclosed in chapters 4 and 5. The observations in chapter 6 can be seen as covered by those two nodes as well. The fifth node, after the first node that has to choices, represents the work explained in chapter 7 where our results are disclosed. The final node represents our discussion and conclusion found in chapters 8 and 9.

The months following August were spent analyzing our results and preparing our conclusions.

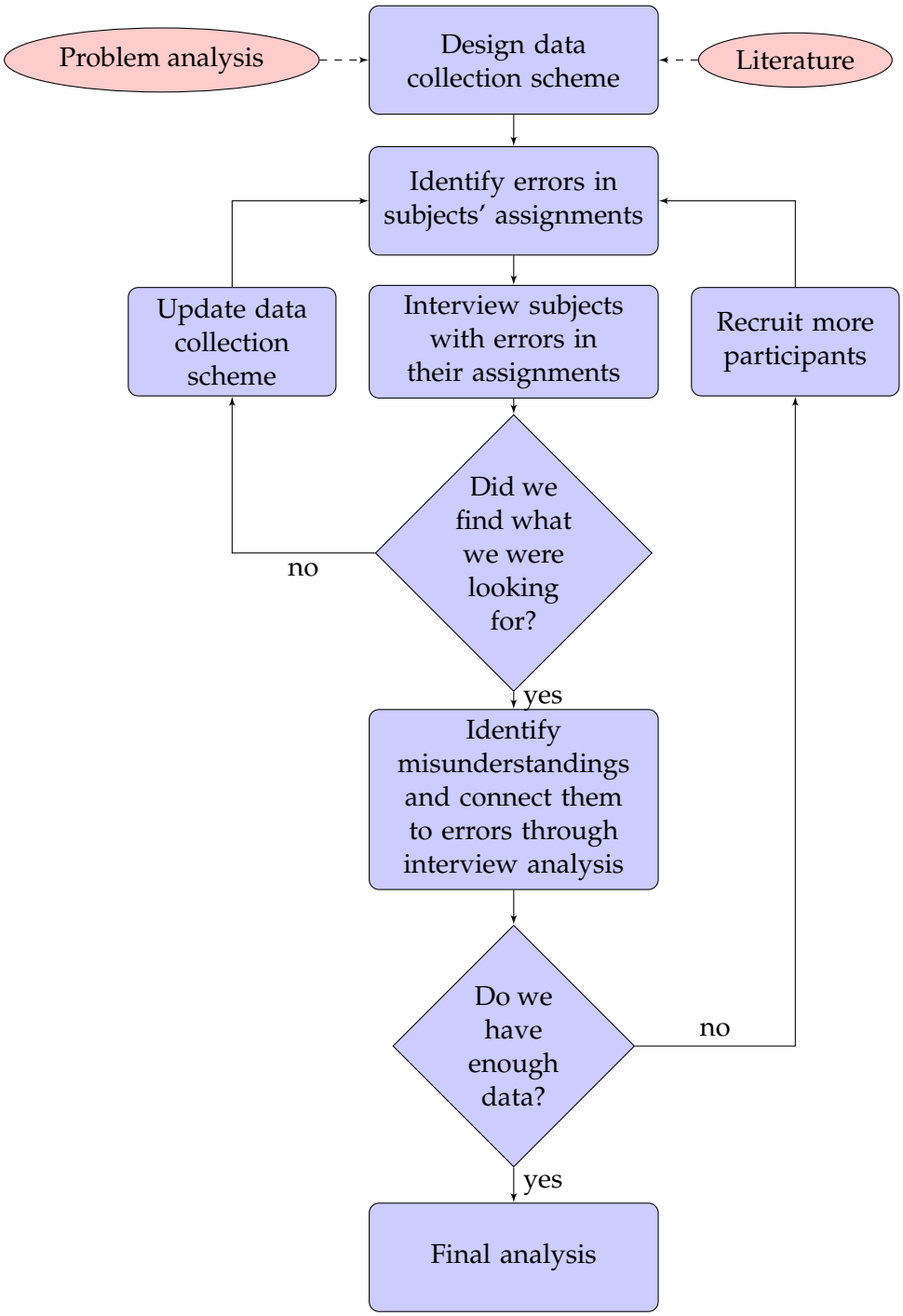


Figure 3.1: Model of our approach to data collection

Chapter 4

Initial interviews

In this chapter we take a look at the first round of interviews we conducted. We will discuss the state of the targeted course at the time the interviews were concluded, the assignment that was reviewed and which errors we expected before the students solutions to the assignments were reviewed. Finally we will discuss the subjects that participated in these particular interviews and compare the errors they made to the ones we expected.

4.1 State of the targeted course

At the time of the initial interviews the students of the targeted course had been through a significant piece of the curriculum. They have been through eight out of 15 lectures that contained new concepts and were not repetition classes.

The assignment solutions reviewed required knowledge of a majority of the intended learning objectives stated in section 2.4.1. The assignment includes subclasses, interfaces, virtual methods, abstract data types and alternative implementations as well as linked lists with the associated algorithms.

Most students solved this assignment using pair programming. But some did not have a partner and solved it on their own. This may have an effect on the results, as it creates an extra reason for why a student may have little knowledge of parts of the code that has been written. We still chose this assignment because lack of knowledge of a particular solution also may give interesting insights into which misunderstandings a student may have.

4.2 The assignment we reviewed

The initial interviews revolved around the student solutions to the sixth mandatory assignment in the targeted course. It is based on the previous assignment. The assignment we reviewed was to make multiple container classes that can hold the classes made in the previous assignments. But none of the errors made in the previous assignment should have an effect

on how they perform in the assignment we review. The assignment text used in the interviews can be found in Appendix E. The assignment text for the previous assignment can be found inn Appendix D.

The assignment we reviewed is made up from multiple parts. Part one is to design and make a drawing of an intended class hierarchy of a specified set of classes and interfaces to be used later in the task. As this study is about students errors in code, we have chosen not to assess that part of the assignment at all.

Note that when we use the word implement we refer to how a class implements an interface and not that a person has implemented a piece of planned code.

The next part of the task was to write the code for the interfaces and classes. The following classes were to be written:

- `AbstraktTabell` : a generic interface for containers where you can add and search for things on given indexes.
- `AbstraktSortertEnkelListe` : a generic interface for sorted containers.
- `Tabell` : a generic container which implements the interface `AbstraktTabell` using an array to store data.
- `SortertEnkelListe` : a sorted generic container which implements the interface `AbstraktSortertEnkelListe` using a linked list structure to store data.
- `EnkelReseptListe` : an unsorted linked list specified to contain a class the students have made in a previous assignment.
- `YngsteFørstResepListe` : a FIFO queue extending the linked list given above
- `EldsteFørstResepListe` : a LIFO queue extending the linked list given above

The final part of the assignment we reviewed was to create unit tests for all the classes made in the second part of the assignment. To keep the amount of work required for the analysis from growing too big we chose to focus mainly on the second task and not spend too much time with the unit testing. It may have been interesting to take a look at how the students designed their tests and find out what,d in their programs, students believed to be important aspects to test.

4.3 Expected student errors

Before the interviews the assignment was analyzed and expected errors were based on a mix of the experience of the research team, interviews with the assignment author, the professors and TAs in the targeted course as well as some literature about student coding errors made at each compilation [1,

6, 33]. The list below do not contain many syntax errors. This is because the students should be experienced enough to deal with all the simplest errors in their code in the three week span they have to complete the assignment.

- No syntax errors were expected
- Semantic errors:
  - A class `implements` another class
  - A class `extends` an interface
  - An interface `implements` another interface
  - A method attempts to initialize a generic array
  - A class or interface does not implement or extending `Iterable` but is attempted used in `for-each` -loops
- Logic errors:
  - An `Iterator` returns null-pointers
  - A class implements `Iterator` instead of `Iterable`
  - A method uses `==` instead of `equals` -method
  - An interfaces or class miss method from assignment description
  - A class does not bind the generic type to required interfaces
  - The insertion-method in an array-container does not handles all edge cases
  - The implementation of an `Iterator` returns `Node` objects from linked list
  - The insertion-method in linked list misuses the `compareTo` -method
  - A method compares generic objects using the result from `toString`
  - The insertion-method in a linked list does not handle all edge cases
  - The concepts of FIFO- and LIFO-queues is mixed
  - The iterator in FIFO- or LIFO-queues outputs values in the wrong sequence
- Design errors:
  - An interface does not bind the generic type to required interfaces
  - A method in an `Iterator` does not throw the appropriate exceptions
  - Next-pointers are used directly in a class used in a linked list
  - The keyword private is used for fields in a super class and the subclass uses `super.variable_name`
  - A class lacks the appropriate heritage from a natural super class
- No design flaws were expected

4.4 The participants

The assignments of ten different students were examined. The assignments of seven of them were interesting. Out of those seven, four were able to make an appointment for an interview. Two of those had cooperated in pair-programming and handed in the same assignment. One of the students who cooperated was clearly an inferior programmer to the other and the work of the better one dominated the assignment. That left little data to be gained from the second as he had little knowledge of what had been done.

The students had a lot more errors in their assignments each than we had anticipated. This may be due to a lower skill level than expected, or that the task simply was more difficult than the level of knowledge expected at that point in the course. But this should not have any negative effects on our data. The misunderstandings of all students of the targeted course that have errors in their code are relevant to our study.

4.5 The errors we experienced

In this section we take a look at the errors we identified in the assignments of the students we talked to in this round of interviews. Most of the errors were identified before the interviews were conducted, but some were missed and identified during the interviews. Any errors that only appeared in the assignments of students who did not participate in the interviews are omitted.

- Syntax errors:
  - Uses equals on a String-object with a generic object as parameter. This is an extended version of using the wrong parameter type.
- Semantic errors:
  - The program contains duplicate classes
- Logic errors:
  - A class or interface `implements` a generic interface without specifying the generic parameter
  - An intended iterable class does not implement `Iterable` interface
  - A method compares generic objects using the result from `toString`
  - A non-recursive insert-method in a sorted linked list contains no loops
  - A non-abstract list-class lacks insertion method
  - A method expands existing storage array but does not update the original array-pointer

  - A linked list contains an extra insertion method for sorted insertion instead of using the inherited insert-method from interface
  - The insertion-method in a linked list fails to properly update pointers

- Design errors:
  - A non-exception class, which is never thrown , extends `RuntimeException`
  - A Node-class in a generic container contains a String id to compare elements
  - A class uses an Object-array for storing generic objects

- Design flaws:
  - A private inner class is declared as generic without any reuse
  - The insertion-method in a linked list uses a loop to insert last element having a `last` -pointer

The students had been programming for at least one semester before they attended the targeted course. The solutions to the assignment were made during a three week period. It is therefore natural, as we predicted in section 4.3, that not many syntax errors were found in the reviewed solutions. In that time the students should have been able to sort out all of the simpler errors causing compiler failure. Yet one of the students did have one semantic error, causing compiler failure, in his code.

We do see that quite a lot of the errors we identified were similar to the ones we expected. We experienced less errors related directly to linked lists, but quite a bit more regarding generic classes and interfaces.

Due to the low amount of participants it is hard to say something definitive about why the errors we experienced were different from the ones we expected. But it seems we underestimated the difficulty of writing and using generic classes and overestimated students issues with linked lists. We also overestimated how many problems the students have with the Java keywords `extends` and `implements` . None of the assignments had any errors regarding these.

Further details about the errors and which misunderstandings we identified during the analysis of the interviews are discussed in chapter 7 and chapter 9.

4.6 Issues and points of improvement

After completing the initial interviews we wanted to answer the following questions. Have we gained any valuable data assisting in answering our research questions? If so, were there parts of the interviews that could be improved, to improve the quality of the data recorded?

We had gathered several interesting insights into the understanding of certain errors and the misunderstandings that caused them. But we had experienced some issues with the interview method that we needed to address in the next round of interviews.

4.6.1 Too long interviews

The interviews became way too long and contains a lot of conversation about things in the assignment that is not relevant to this study. This does not affect our data, but causes a lot of extra work during the analysis that may have been avoided through more careful planning of the interview.

4.6.2 Errors that escaped our review

We identified some additional errors during the interviews, as we had overlooked them when first reviewing the assignments. Our hope is that this issue may be fixed if we spend a bit more time running the programs instead of simply reading them. Then we can cut some parts of the assignment and not have the students talk about everything.

4.6.3 Too much time elapsed before the interviews

We believe the students may have forgotten some of the specifics considering their assignments. Some of the interviews were held as much as ten days after the assignment deadline, and the time frame of the assignment was three weeks. We had to wait for the deadline to elapse before any of the assignments were made available even though some students may have handed in their solution long before the deadline.

Some of the tasks may have been solved as much as 30 days before the interview. This may cause us to miss some misunderstandings that we could have caught had we talked to the students earlier. Some of the students may have learned what they did wrong and no longer have the misunderstandings that caused certain errors.

4.6.4 The assignment was too long

The assignment was a bit too long to effectively conduct many reviews and interviews. The different Java classes that the students were to write were often not related to each other, making the reviews difficult and adding to the interview lengths.

Chapter 5

Second round of interviews

In this chapter we take a look at the second round of interviews we conducted. We will discuss some experiences that was made during the initial interviews and how they affected how the second round of interviews were conducted. We will then take a look at the assignment that was reviewed and which errors we expected before the students solutions to the assignments were reviewed. Finally we will discuss the subjects that participated in these particular interviews and compare the errors they made to the ones we expected.

5.1 Experience from the initial interviews

The initial interviews proved that our method of gaining insights into which errors students make in their solution to instruments of assessment worked. For that reason no major changes were made. Some minor changes to the interview technique was made in an attempt to rectify some of the issues listed in section 4.6. The issues we tried to deal with was the overly long interviews and time lapsed between when a student solved a problem and the interview was conducted. The following steps to improve our results was made:

- We worked to become more concise in our questions and try to make the interviews shorter than in the initial interviews.
- We tried to get the students to participate within a shorter time frame of their deadline to attempt to reduce the time between a student solved a problem and the interview was conducted.
- The assignment that was chosen only spanned two instead of three weeks, reducing the time between when the assignment started and was handed in.
- We asked the students who had signed up for the study to email us as soon as they handed in the assignment allowing us to review the it earlier than in the initial interviews.

- We chose to ask even more questions about the errors we had identified in students assignments. This includes getting a full explanation of the concept and some of its constituent parts.

5.2 State of the targeted course

At this point the students of the targeted course had been through all the new concepts in the course excluding graphic user interfaces. For that reason we can expect the assignment reviewed in the interviews to require knowledge of any of the concepts stated in the intended learning objectives from section 2.4.1 on page 12.

The assignment requires knowledge of subclasses, virtual methods, abstract data types and alternative implementations as well as the possibility of linked lists with the associated algorithms. What makes it more difficult than some previous assignments is that it requires more from the students in regards to planning and designing their program.

Unlike the assignment reviewed in the initial interviews everyone had to complete this assignment on their own. Pair programming was not allowed.

5.3 The assignment we reviewed

The initial interviews revolved around the student solutions to the ninth mandatory assignment in the targeted course. The assignment text can be found in Appendix F.

The assignment is to make the first step in creating a program able to solve Sudoku boards using a brute force method. Sudoku is a logic-based number puzzle[55].

The task is split into three assignments. In the first assignment the students are asked to create the underlying data structure of a Sudoku board. That includes classes and structures to represent squares, rows, columns and boxes. In addition they need to write a file reading scheme for text files containing the dimensions and values of a given Sudoku board as well as a class to store valid solutions to the Sudoku board.

The second and third assignments involve creating a recursive method to try all possible solutions of the board and a graphical user interface for the Sudoku solving program.

5.4 Expected student errors

Before the interviews the assignment was analyzed and expected errors were based on the same sources as the initial interviews (see section 4.3 on page 34). The difference in the analysis of the assignment as is that we used some of our experience from the initial interviews. The design errors and flaws may be hard to understand unless you read the assignment text.

- No syntax errors were expected

- Semantic errors:
  - The program does not handle input files with alphanumerical values
- Logic errors:
  - Squares reference the wrong Box-objects
  - Squares lacks linked list structure
- Design errors:
  - Use of ArrayList instead of array for storage when indexes and number of elements are known
  - Almost identical classes are defined without using heritage from a super class
  - An object use "tags" to define which other objects it "belongs" to instead of being a pointed to by a field in the other objects
- Design flaws:
  - A class uses two-dimensional arrays for storage of elements when position is irrelevant
  - An ArrayList is used instead of an array when maximum length is known but final size is uncertain

For this assignment we expected more design flaws than the one reviewed in the initial interviews. This is partly because the assignment requires an unfamiliar yet quite strict data structure. The chance of students making poor design choices was considered as quite high by the teaching staff in the targeted course.

We also expected less semantic and logic errors than in the assignment reviewed in the initial interviews. This is because of the smaller size of the assignment as well as a correction based on our experiences from the initial interviews.

5.5 The participants

We reviewed 23 different assignment solutions and interviewed ten students about them. Another seven were invited to participate but either did not respond or were unable to make time for an appointment. The final six did not have enough interesting errors in their code to be invited. The errors identified for the seven who were unable to make time for an appointment did not show a different pattern than for those who were interviewed. They would not have added any errors to our list but their misunderstandings may have been different.

In the initial interviews most of the students that were interviewed made a lot more errors than we had expected. This time around it was a lot less. The added participants this round had less errors and showed an

all over better understanding of the assignment and their work. This may be because they were better programmers, or because the assignment was easier than the last one.

The very best students were not invited to participate in the interviews as their solutions did not contain sufficient errors to be interesting for this study.

5.6 The errors we experienced

In this section we take a look at the errors we identified in the solutions of the students we talked to in this round of interviews. Most of the errors were identified before the interviews were conducted, but some were missed and identified during the interviews. Any errors that only appeared in the solutions of students who did not participate in the interviews are omitted.

- Logic errors:
  - A method throws or catches the `Exception` super class
  - Program failure data is stored outside the thrown exception
  - A method throws wrong exceptions at failure events
- Design errors:
  - An ArrayList is used instead of an array for storage when indexes and number of elements are known
  - Almost identical classes are defined without using heritage from a super class
  - An object use "tags" to define which other objects it "belongs" to instead of being a pointed to by a field in the other objects
  - The program contains extra data structures to describe the state of the program state instead of using appropriate methods to compute the state
  - A non-exception class extends an exception
  - The program does not convert the data type of values read from a file to appropriate non- `String` types
- Design flaws:
  - An ArrayList is used instead of an array when maximum length is known but final size is uncertain
  - A class uses two-dimensional arrays for storage of elements when position is irrelevant
  - The program contains improper use of private internal classes
  - Functionality is located in another class than the one it should be in

- A class uses different data types for storage of data and functions used on those data and converts between the two types

Most of the students managed to make their program work properly, so a lot of the errors are classified as design errors and design flaws and lack of attention to the object oriented model. We did not experience a single syntax or semantic error. All programs compiled with no issues. Some errors are classified as logic errors even though the programs produced the correct result. That is because they are related to exceptions and error handling which may cause the wrong result to be produced in particular situations and uses.

Further details about the errors and which misunderstandings we identified during the analysis of the interviews are discussed in chapter 7 and chapter 9.

5.7 Issues and points of improvement

The second round interviews were a lot more to the point in comparison to the initial ones. This, combined with the fact that the task was smaller made both the interviews and the analysis a lot easier. Our attempts at addressing the issues from the initial interviews were successful on most points. We spent more time analyzing and reviewing the assignment solution before the interviews. This made analyzing our results a lot easier.

The second round of interviews gave a lot more relevant data than the initial interviews. After completing the interviews we only have a small list of possible improvements.

5.7.1 The time elapsed could have been improved further

Similarly to the initial interviews a lot of students had forgot some of the reasoning behind their choices. Some students had also corrected the errors and problems they had as the code they had made were to be used in a new assignment. Because of that they were sometimes aware of the problem but had already replaced their faulty knowledge with new understanding rendering the information we sought lost forever. If possible we would have conducted the interviews as soon as the day after the solution was handed in to improve upon this measure.

5.7.2 The assignment may have been a bit too easy

This assignment was far more fitting in size than the assignment used in the initial interviews. The assignment may have been a bit too easy as almost all the students managed to complete it without any errors causing the program to produce the wrong result.

Chapter 6

Think-aloud observations

In this chapter we take a look at the observations conducted to gather supplemental data to the assignment reviews and interviews. This final data collection activity was an observation of subjects programming small exercises. We will discuss the state of the course when the observations were conducted and the subjects we expected to participate in the observations. We will then take a look at the exercises that were made for the observations, how the topics were chosen and which errors and misunderstandings we hoped they would add further data to. Finally we will take a look at the subjects who actually participated and what data the observations contributed to.

6.1 State of the targeted course

At the time of the observations the semester was over. The final exam was finished and all students who got a passing grade on the final exam had completed the targeted course. The observations were held as part of an extra set of lectures and problem solving sessions, offered to students who were allowed to resit a the exam in August. The reasons for having to resit the exam are usually that students either get a failing grade on the previous exam or fail to show up due to being ill.

For this reason we expected the subjects in this group to have a skill level well below average for the targeted course. A large portion of them did not manage to get a passing grade on the previous exam. For that reason they are not truly a representative group for the course as a whole. It is important to know that when considering the data gained from these observations.

6.2 The exercises

The goal of the exercises was to supplement the data we collected in the interviews. This was done to help us identify more misunderstandings causing the type of errors we previously had experienced. Therefore the exercises had to be designed to suit our previous results. That gave us the

following list of possible error types to look into:

- Exception errors
- Poor choice of data structure
- Data and functionality distribution errors
- Linked list errors
- Array-errors
- Generics errors
- Iterator errors

6.2.1 Limitations from the think-aloud method

Because the exercises were to be solved during think-aloud observations it was important to design the exercises to fit the observations with the method described in section 3.5.1.

With regards to the work required to analyze the observations and the time available to the students who were practicing to resit the exam the exercises should not take more than 30 minutes to complete for a skilled novice programmer.

The topics covered should be ones the subjects do not master fully yet manage to at least attempt creating a solution to. If the exercises are too hard no subject will be able to make a proper attempt. If the exercises are too easy they will have learned how to solve it using rote learning and a lot of the choices made would be unconscious ones. This is a reason why it is often a problem to figure out how experts solve simpler problems. They have reached a point where they know it so well they no longer have a good way of verbalizing the process they went through to solve the problem.

Additionally there should be a warm-up exercise to let the subjects get comfortable with verbalizing their thoughts and actions when they write program code. That exercise should take five to ten minutes and not be included in the recording of the observation to help the subjects relax and get comfortable.

6.2.2 Limitations due to the subjects' skill levels

As the subjects were expected to not be the most skilled programmers the exercises would need to consist the simpler parts of the core curriculum. We had no other data from the study to help us do this. But we were allowed to use the scores achieved in the final exam in the targeted course held a few months earlier. One of the professors teaching the targeted course provided us with the scores of the students who failed the exam. We were given the scores of the individual tasks of the exam without the candidate numbers of the students.

Most of the students we expected to participate in the observation would have tried to solve that exam and gotten a failing grade. Therefore it

	Theme
Task 1	Class hierarchy and pointer structure
Task 2	Data structure from program definition
Task 3	Ordered linked list and recursion
Task 4	GUI, Java Swing and Java AWT
Task 5	Threads, monitors and sorting algorithms

Table 6.1: INF1010 June 2015 exam overview

	$\bar{x}$	$\tilde{x}$	s	Students with 0%
Task 1	20.82%	10.00	23.89	23
Task 2	23.65%	17.50	24.79	10
Task 3	19.82%	18.50	14.36	6
Task 4	20,49%	11.25	19.32	14
Task 5	9,03%	5.50	10.67	35

Table 6.2: Statistical analysis of the INF1010 June 2015 exam scores for students with failing grades

helped us make some assumptions about which topics are suited for them to solve during an observation. If they scored too high in a topic it might be too easy. If they scored close to zero it is unlikely that asking them to solve such an exercise may give any insights into their thinking, as they will probably not have any idea at all on how to solve it.

The exam of June 2015

The exam was made up of 5 different tasks. Those may also have been divided into extra subtasks. But the scores we were supplied with were separated into those scores only. Therefore we have chosen to consider them as a whole. The topics of the different tasks can be found in Table 6.1 and a histogram containing the distribution of the students' scores can be found in 6.1 on the next page.

To analyze the scores the students achieved we made a quick statistical analysis. We used the statistical measures we presented in section 3.7 on the students' scores. The results of those measures are listed in Table 6.2.

The statistical measure mode is not included as it computed to 0 for all five tasks. When the granularity of the scoring system is so detailed (0-100%), the number of equal non-zero scores are unlikely to be very high. Instead the number of students who did not get any points at all were added as a metric. This is very relevant as it may highlight topics that should be avoided in tasks in the observation.

The mean, $\bar{x}$, is quite similar, at approximately 20%, for all tasks except task 5 which is at 9%. So it is hard to make any assumptions except that threads, monitors and sorting algorithms would have been likely to be too difficult to fit in the exercises for the observations.

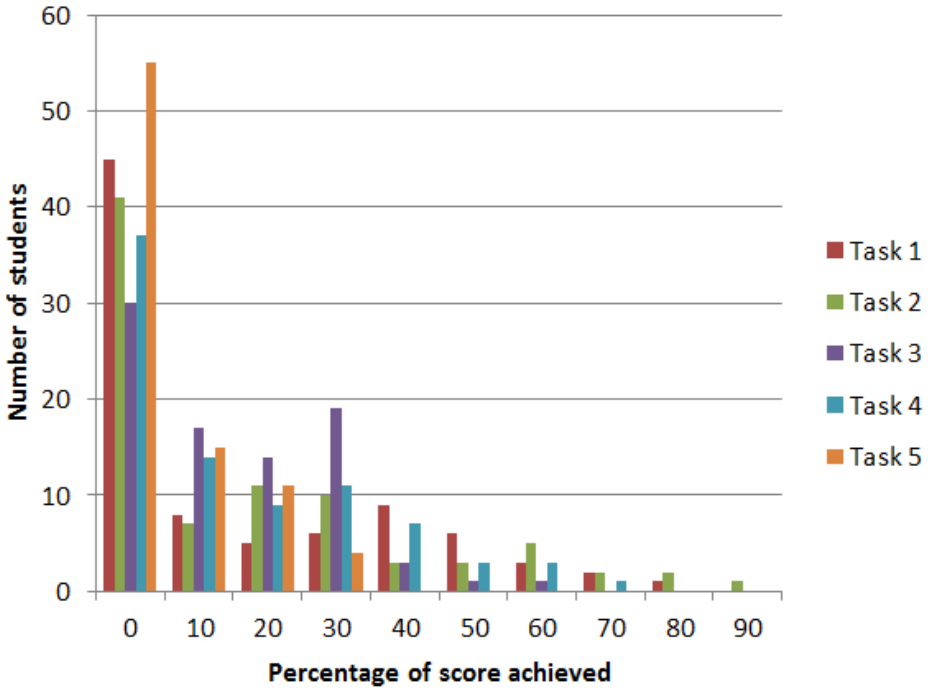


Figure 6.1: The distribution of students failing the INF1010 June 2015 exam based on percentage of the maximum score they achieved

The median, $\tilde{x}$, on the other hand varies a bit more. A median lower than the mean indicates a large amount of students scoring very low and a few students scoring very high. A median above the mean indicates the opposite. And a median closer to the mean indicates a more even spread. A median above the mean indicates the opposite.

Finally a low standard deviation, s, and few students scoring 0% shows less spread in the score and more students who got a score close to the mean as the mean cannot have been pulled up far by a few very high scores.

From this we can see that the third topic, linked lists and recursion, may prove to be a good choice for these students.

6.2.3 The chosen set of exercises

With all the aforementioned requirements in mind we designed the set of exercises described below. The full text given to the subjects can be found in Appendix G.

- A warm-up exercise requiring the subjects to create a simple print-method for a class called Car and create a functioning program that creates a few instances of the class Car and prints the info about them.
- The first proper exercise required the subjects to design and program a simple class hierarchy for a vehicles with some predefined requirements.

- The second exercise required the subjects to program a LIFO-queue in the form of a linked list.
- If the subjects were to complete both exercises the subjects were to solve an additional exercise to make the LIFO-queue from the second exercises iterable.

How will these exercises fit our requirements?

The warm-up exercise fills the requirement to ease the students into verbalizing their thoughts.

The first proper exercise is designed to help us identify misunderstandings that lead to errors in distribution of data and functionality. Those were very common errors in the interviews. It was made to be a lot easier than the exercises from the final exam from June 2015 that consider class hierarchy and pointer structure. So it should be feasible to solve for the students participating.

The second exercise was designed to help us identify misunderstandings that lead to errors with linked lists and generic classes. They are two types of error we experienced a bit, and the teaching assistants in the targeted course described them as "very common" in our talks with them.

The additional exercise was made to help us identify misunderstandings that lead to errors both in linked lists and iterators. This is achieved because writing an iterator requires knowledge of the container is written for.

We did not have a student with the proper level of experience available to test our exercises. Instead they were tested by a student with one additional year of experience. He spent twenty-six minutes solving the exercises. From this we believe a skilled student with one year less experience may have to spend a couple of minutes more.

6.3 The participants

27 students signed up for the extra lectures and problem solving session. They were offered a crash course consisting classroom lectures and plenary problem solving. 16 out of the 27 showed up for the crash course and of them a total of nine students chose to participate in the individual observations.

The lectures were held over two days and the individual observations were held afterwards. The lectures were not designed to improve skills in topics specific to the observations but to prepare them to resit the exam. Thus the topics covered in the observation exercises did not receive any more focus than other topics in the course.

The nine participants were all students who were going to resit the exam. As they chose to participate themselves there was no random selection of students. Based on their choice to participate first in the crash course and then the observations we can expect them to be among the more motivated students. But as mentioned in the beginning of this chapter,

most of them are still likely to be less skilled than the average student of the targeted course.

6.4 Supplemental data collected

The observations were focused on dealing with the student misunderstandings we wanted to learn more about, but this chapter deal only with which errors we have supplemental data to. By that we mean which errors from the interviews that the observations yielded more misunderstandings. We summarize the results from the observations as a list that only contains errors that we have identified in the assignments reviewed for the interviews that we have found supplemental data for.

- Logic errors:
  - A class or interface `implements` a generic interface without specifying the generic parameter
  - A method compares generic objects using the result from `toString`
  - The insertion-method in a linked list fails to properly update pointers
- Design errors:
  - Almost identical classes are defined without using heritage from a super class
- Design flaws:
  - A private inner class is declared as generic
  - The insertion-method in a linked list uses a loop to insert last element having a `last` -pointer

But similarly to most research that look at errors made while novices are programming and attempting to compile their programs, we observed a lot of syntax and semantic errors[1, 6, 7, 33]. But we had not identified many of them in the solutions to assignments reviewed before the interviews. Therefore they are not represented in the list above.

There are fewer errors represented here than in chapter 4 and chapter 5. This is caused by the shortened format of only 30 minutes to write the code and the fact that we omitted all errors that did not occur in the solutions we reviewed. "Fresh" errors from the observations may not have been present in a final delivery with a longer time span than 30 minutes

In addition to insights about the errors listed above, the data from the observations may lead to useful insights into other misunderstandings that do not consider the errors that the subjects made during observations. Further details about these errors and which misunderstandings we identified during the analysis of the interviews are discussed in chapter 7 and chapter 9.

6.5 Issues and points of improvement

After completing the observations the following points are things that either were or could have been addressed to improve the data recorded.

6.5.1 Method

Some of the subjects fell back into habit at times, stopping to verbalize their thoughts. We were not always quick enough to address this and have them back on track.

6.5.2 The subjects

Some of the subjects were not the best suited for observations using the think aloud method. About a third of the subjects needed a lot of reminders to continue speaking. One had trouble typing and speaking at the same time. And a few had issues with keeping their comments free of references to visual cues. The using of visual cues makes the analysis very difficult. The observer is forced to be a lot more attentive and to make good notes to be sure he knows what the subject is referencing.

This may have been improved by having more subjects so the less suitable ones could be replaced. The other problems could have been solved by making it clearer that their comments should be verbose enough to describe the thing they are referring to without a doubt.

6.5.3 The exercises

The subjects differed quite a lot in how much of the exercises they managed to complete. Two of the subjects never got to start on the second proper exercise, whereas one of them did not finish the first. The rest completed the first but only three of them completed the second one. None of them had no errors in their solution of the second exercise. Only one subject started to solve the additional exercise but did not have enough time to complete it.

After observing the first two subjects we learned that the first exercise was a bit too long, with much of repetition without giving any new insights into the subjects abilities or knowledge. It was fixed before the last seven observations with some minor changes to the exercise. A few required classes were removed and we made sure to make a better job at telling the subjects when they could move on to the next exercise.

The second exercise may have been a bit unclear in the formulation of the requirement that the class parameter should be unrestricted. It was formulated so that the class should be a container able to store any object. A lot of subjects interpreted this as a requirement that a single instance of the container class should be able to contain any of the classes in Java at once.

The second exercise could also have been improved by revising the formulation of the specifications of the `remove` method.

6.5.4 Tools

In retrospect it would probably have been useful to record the computer screen as the subjects were programming. It would have made visual references easier to understand and be better than having only the final code when reviewing the work of the subjects.

Chapter 7

Results

In this chapter we go further into the results from the interviews and observations. We will share our findings of different misunderstandings students of the targeted course seem to have and how these misunderstandings cause the errors we have seen. Some readers may need additional explanation of some of the Java constructs discussed in this chapter, such as specific classes. The complete definitions of all classes and interfaces can be found in Oracle's Java 8 API[11].

To make the reading of this chapter easier, we start with a short explanation of how we deal with separate errors and misunderstandings. Section 7.1 contains an explanation on some of the error types and specific errors we have chosen to not focus on. Section 7.2 to 7.7 each deals with a different type of error. The types are created based on subject. For each type of error we first define what requirements an error must fit to be that kind of error. Then we list the different errors we encountered, their classification based on the error classifications in section 2.2.2 and whether they are discussed in depth in this chapter. The most interesting errors are discussed in depth as we:

1. Explain how the error can be recognized and explain using code examples from the assignments we reviewed[1]

2. Identify the students' misunderstandings by taking a look at their statements[2] and sometimes comments from their code

The misunderstandings we identify are categorized in chapter 8, where we will discuss the different misunderstandings we have encountered, how they led to the different types of errors and what may have caused them.

As the assignments we reviewed in the two rounds of interviews were about different topics the amount of overlap of data between the two is not great. But in some classes of errors we do find examples from both rounds.

[1] *The code examples have been edited and stylized to fit the pages and ensure anonymity for the subjects.*

[2] *All original quotes are in Norwegian but have been translated to English. As our study does not concern itself with the gender of the students all of them are referred to as 'he'.*

7.1 Omitted errors

In this study we have identified a large amount of different errors in the assignment solutions of students that would have been possible to discuss in depth. From all the errors, we have chosen the ones we believe are the most important to discuss in this chapter. A number of errors are omitted from further study based on a few different reasons that we have listed below. All errors that are omitted qualify for at least one of those criteria.

- Some errors have *too few occurrences* in the solutions of the students we interviewed. In these cases we often believe the errors to be less common among students of the targeted course and outliers compared to the other errors. Cases that may contradict this are: Either the teaching assistants or professors in the targeted course have specified the error as common. The error is present i multiple solutions of students we did not get to interview.

- Some errors yielded too few revealing statements about the misunderstandings that may have caused them. This causes us to have *too little data* to comfortably make conclusions about their analysis.

- Some errors are *less relevant* to the topics in the targeted course. We avoid focusing our analysis on these errors as we believe they are less likely to be caused by misunderstandings that are highly relevant to our study. We wish to focus on mainly identifying misunderstandings about topics the students are supposed to have learned.

- Some errors yielded mainly misunderstandings that are *covered in the analysis of other errors*. We avoid spending time on multiple errors that do not add any new errors. Those errors will be mentioned in the analysis of the errors that are focused on, having the same misunderstandings .

Most of the errors we identified are listed in the following sections. But some errors, such as errors concerning iterators, were categorized as types that only contained errors we omitted. Those categories and their errors are left out of this chapter entirely.

7.1.1 Most design flaws are omitted

As a result of the third of the criteria above a lot of the design flaws are omitted. We can see why many of them fulfill the last criteria listed above in the definition of design flaws from section 2.2.2:

> A design flaw is an error that arise from bad choices considering program efficiency, scalability and robustness, that are outside the scope of the course but more experienced programmers would avoid.

The fact that they are outside the scope of the course cause them to be loosely linked to the topics taught by the targeted course and are far less interesting to us. However, we will still take a look at some of the data gathered about the errors that are omitted, as some of the errors we highlight are closely connected to these flaws through the misunderstandings that have caused them.

Error	Category	Further analysis?
A method throws or catches the `Exception` super class	Logic error	This error is analyzed further in section 7.2.1
Failure data is stored outside exception	Logic error	This error had *too few occurrences* and is not considered common.
A method throws wrong exceptions at failure events	Logic error	This error had *too few occurrences* and is not considered common.
A non-exception class, which is never thrown , extends `RuntimeException`	Design error	This error had *too few occurrences* and is not considered common.

Table 7.1: Errors and flaws we identified that considers exceptions

7.2 Exception errors

These are errors that arise from poor use of exceptions. To be able to consider these errors properly we first take a look at what an exception is.

Put in a pure technical view exceptions in Java are objects with the special ability of being `Throwable` . That means that they can be passed to the calling function in a different way than using the return value, giving the programmer an additional way of handling errors. According to the textbook of the targeted course, *Big Java: Late Objects*[32], an exception is:

> "A class that signals a condition that prevents the program from continuing normally. When such a condition occurs, an object of the exception class is thrown."
> (Horstmann, p. 958, [32])

Instead of looking for a predefined "error-value" in the return value the programmer uses a special keyword `catch` to get hold of an exception object. Exceptions are divided into different classes. Typically the class of an exception decides the type of error that arose. The object may contain additional information such as where in the code the issue appeared and which method calls that led to it - called stack trace.

The different errors we encountered considering exceptions can be found in Table 7.1.

7.2.1 A method throws or catches the Exception super class

The most common error concerning exceptions was the use of the statement `catch(Exception e)` . This statement catches all possible exceptions. It may hide both minor and major flaws in a program causing the

```
catch (Exception e) {
    System.out.println(
        "The file: " + filename + " was not found");
    return;
}
```

Figure 7.1: `catch-block` used to handle a single subclass of the `Exception` super class.
(*From the code reviews for the second round of interviews*)

programmer to never realize an issue appears. In the worst case a program catches an exception that is produced very rarely. If the exception is not handled properly and information is stored, fixing the problem will not be possible until the next time it arises. This is not good in critical systems.

In addition it makes the program less readable as the reader have no easy way of gaining insights into which exceptions the `catch` -block was intended to handle.

The error is in some cases less critical as the students include the statement `e.printStackTrace()` , or something similar, in the `catch` block. That statement prints any types of exception to the terminal if `e` is an `Exception` -pointer. However, the readability of the code is still impaired. In multiple cases the interviewer was required to explicitly change the `catch` block to handle a `FileNotFoundException` to ensure no other exceptions were meant to be handled.

This error is a bit special as we have split it into three different types of occurrences to help us understand the motives the students have when using the `Exception` super class:

- The `Exception` class was used to catch one specific exception, as shown is Figure 7.1.
- The `Exception` class was used to catch multiple exceptions in a single try-catch statement.
- The `Exception` class was used to catch instances of the class `Exception` , as shown in Figure 7.2 on the next page

The Exception class was used to catch one specific exception

This case was the most common of the three different cases. Most of the students explained the use of `catch(Exception e)` by saying they needed to catch an exception or the program would not have compiled. We will see why they did not choose a specific exception when looking at specific misunderstandings that may have caused this error.

```
try{
    ...
    if (counter > data.length)
        throw new ArrayIndexOutOfBoundsException();
    else if (counter < data.length)
        throw new Exception();

    createNewBoard();
}
catch (FileNotFoundException e){...}
catch (NumberFormatException e){...}
catch (ArrayIndexOutOfBoundsException e){...}
catch (NullPointerException e){...}
catch (Exception e){
    System.out.format("\n"
            + "[ERROR]: file format\n"
            + "The two frist lines specifies a "
            + "%d x %d board with %d squares.\n"
            + "readFile() has not found enough data "
            + "for %d squares in your file \"%s\".\n"
            + "\n",
            + row, col, n * n, n * n, filename);
}
```

Figure 7.2: `catch` block used to handle instances of the `Exception` super class.
(*From the code reviews for the second round of interviews*)

The Exception class was used to catch multiple exceptions in a single try-catch statement

One student had this error in multiple places in his code. When asked about it he had similar arguments as the students that showed the first behavior. His explanation to the use was that it handled the exceptions he wanted to catch and let the program react accordingly. When asked about other possibilities to solve multiple exceptions in a single `try` -statement he had no alternatives in mind. His answers did not make it apparent that he knew of the possibility to use multiple `catch` -statements. When informed of the possibility he still preferred his own solution as it had fewer lines of code.

The Exception class was used to catch instances of the class Exception

This last error only appeared once and we do not believe it to be very common. The code in Figure 7.2 contains far more than one error, but we will concentrate on this one. The explanation differs as the student had chosen `Exception` to represent a specific error incident. He chose to use it because he thought he had no other exceptions available to him. His words exactly were:

> *" It was the only one I hadn't used "*
> (participant, second round of interviews)

He also states that he finds handling large bodies of text inside the program logic quite messy. So he preferred moving the text-generation into the `catch` blocks instead of adding it as a message in the instances of the exceptions.

Encountered misunderstandings

Despite some minor differences in explanation we believe that there are a few common misunderstandings between most of the occurrences of this error. We have identified three areas where the students understanding is lacking and examine them a bit more in detail. Each student lacks proper understanding of at least of one these.

Exception type is not important when handling exceptions

The students show little understanding of the hierarchy of exceptions and how exception type in many cases determine what went wrong in the program. One student summarizes his knowledge of exceptions by limiting it to the fact that he knows that there exists multiple ones but he has not learned them:

> *" I've seen the [exception] class hierarchy... or tree. But I haven't bothered learning about it. It did not seem very important when they showed it to us in class. "*
> (participant, second round of interviews)

This shows a lack of knowledge of just how important the type of exception may be.

Some students did not show any understanding that catching the `Exception` super class provides any potential issue to be aware of. When asked about how it affected their program their answers reflected that it made no difference at all. The exception they wanted to catch was handled properly. One student explained that he used `Exception` simply as he would get a compiler error unless he did so:

> *" Because of the `Scanner` I get an error message if I don't catch an exception. "*
> (participant, second round of interviews)

Once the compiler error no longer showed up the student considered the problem to be solved.

This seem to be consistent with the findings of Rashkovits and Lavy[46]. In their study to pinpoint college students level of understanding of exceptions, only 57.5% reached a level of understanding where:

> "... the student understands that the non-hierarchically-related thrown exceptions can be caught and handled inside the calling method in separate catch-clauses referring to these exceptions. She also understands that a try-catch block can contain one or more commands, and when an exception is thrown the fluent execution stops and the control is passed to the appropriate catch-clause and continues from there ..."
> (Rashkovits and Lavy[46])

As explained in section 3.8 the students participating in this study do not fit the average skill level of students attending the targeted course. We may expect most of them to belong to the less skilled half of students. Therefore it is very likely that even more than 40% of the participants in our study have trouble reaching that level of understanding.

That does not mean none of our participants were aware that there is a difference. One student showed he was aware that he should have used a different exception.

> *" Here I have a catch-clause with `Exception`. It should have been a `FileNotFoundException` but I did not remember which package to include it from. "*
> (participant, second round of interviews)

When asked why it should have been a `FileNotFoundException` his answer was that being explicit when catching methods is good practice. While not showing perfect understanding of why it is good practice he was aware that it was an issue, unlike many of the other students.

Exceptions do not adhere to the same rules a other Java-classes

The fact that the student quoted above hesitates to call the class hierarchy of exceptions a class hierarchy and chooses to change to call it a tree made us believe the student is not completely aware that exceptions are defined just as any other class in Java. This is consistent with the statements of the student who used `Exception` to model a specific error incident instead of writing his own code. When asked why he had not done so he answered that he was not aware that was possible. Yet the possibility to extend a class with a subclasses is a common trait for all classes in Java.

From our talks not all students showed awareness of the fact that `catch(Exception)` will catch any runtime exception, such as `NullPointerException` and `IndexOutOfBoundsException`. This supports the lack of understanding how heritage affects all exceptions as any other class. This may also be related to a lack of understanding how handling a super class or interface make a method, pointer or `catch` -statement deal with all subsequent subclasses. It may also be that they do not really understand that exceptions they need to catch explicitly do not differ from exceptions that are handled implicitly by the Java Virtual Machine.

Program robustness is not as important as other program attributes

Another first thing the students fail to explain is how their solution impact program robustness. They talk as if their program is finished and will not ever have any work made to expand upon it again. Even though that is true in some cases students should still learn to code as if there always will be changes made. They may need to implement such changes themselves, when they have forgotten their intentions. Or someone else may be tasked to do so.

When asked if catching `Exception` could cause a problem a few students mentioned that no other exceptions appeared in their program. One said the following:

> *" That's not a problem. The exception from the `Scanner` is the only one which can appear. "*
> (participant, second round of interviews)

The students with this kind of explanation shows signs of understanding that multiple exceptions may appear. But their view of their program seem somewhat static and finished.

The students seem to think the most important thing with exceptions was to catch them so the program would compile. Potential issues at runtime was not something they really got into when asked to elaborate on their choice to catch the `Exception` super class. They seemed happy their program already produced the correct result.

IDE solutions to problems are good enough

Another student had obviously used an IDE that had solved the problem of handling the exception for him. We can see from the comment in his code,

```
} catch (Exception e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

Figure 7.3: Auto-generated `catch` block made by an IDE
(*From the code reviews for the second round of interviews*)

given in Figure 7.3, that this is the work of an IDE.

But the code in the `try` block only produced one exception that was not a `RuntimeException`, namely `FileNotFoundException`. This may have been caused by blind trust that the IDE would solve the issue in a good way. Even when this student's answer to whether it was a problem was similar to that of the others we believe IDEs may at least cause these kind of errors to appear more often.

Error	Category	Further analysis?
The insertion-method in a linked list fails to properly update pointers	Logic error	This error is reviewed in section 7.3.1
The insertion-method in a linked list uses a loop to insert last element having a `last` -pointer	Design flaw	We consider this error to be *less relevant* to the topics in the targeted course.
A non-recursive insert-method in a sorted linked list contains no loops	Logic error	The interviews and observations yielded *too little data* about this error.
A non-abstract list-class lacks insertion method	Logic error	This error had *too few occurrences* and is not considered common.
A linked list contains an extra insertion method for sorted insertion instead of using the inherited insert-method from interface	Logic error	This error had *too few occurrences* and is not considered common.

Table 7.2: Errors and flaws we identified that considers linked lists

7.3 Linked list errors

These are errors that arise when implementing linked lists. According to the textbook in the course a linked list is:

> "A data structure that can hold an arbitrary number of objects, each of which is stored in a linked list object, which contains a pointer to the next link"
> (Horstmann[32])

This means that linked lists require a steady mind when updating pointers to ensure the links are organized in the way you want. The errors we experienced with linked lists, their classifications and whether we will go further into them in this chapter are shown in Table 7.2.

7.3.1 The insertion-method in a linked list fails to properly update pointers

This error concerns insertion and deletion from linked lists. Typically this is due to bad updating of the `first` - and/or `last` -pointers in the

```
public void insert(Prescription p){
    Node temp = new Node(p);
    nrOfElements++;
    if (first == null){
        first = temp;
        last = temp;
    }else{
        /* We put the new prescription in the
           start of the list - meaning last.next
           points at the oldest prescription */
        last = first;
        temp.next = last;
        first = temp;
    }

}
```

Figure 7.4: Pointer error in insertion-method in a LIFO-queue implemented using a linked list structure.
(*From the code reviews for the initial interviews*)

linked list. This may cause that the element is lost forever or that the list is compromised losing other data or causing runtime errors when other methods are called.

The code in Figure 7.4 and Figure 7.5 show two methods that attempt to do insertion in a LIFO-queue.

The code in the first figure attempts insertion in the beginning of a linked list. But what really happens is that `first` does end up pointing to the newest element of the queue but the `last` -pointer ends up pointing to the second one instead of the last one.

The second method attempts to insert the new element in the end of

```
    public void insert(Node next) {
        if (head == null) {
            head = next = tail;
        } else {
            Node temp = tail;
            tail = next;
            temp.next = tail;
        }
    }
```

Figure 7.5: Pointer error in insertion-method in a LIFO linked list where the new element is lost.
(*Code written during the observations*)

the list instead. But it has an error when inserting the first element in the queue. Both `head` and `next` are set to `null` as the assignment of pointer values appear in the wrong sequence. The list `head` and `tail` remains `null` after the `insert` -method is done.

Encountered misunderstandings

In the interviews only one student whom we spoke to delivered this type of error. But linked list pointer issues appeared in more of the other assignments we reviewed where we were unable to interview the students. Additionally this was one of the issues that was highlighted in all the interviews with teaching assistants in the targeted course.

Data structure visualization is too difficult

When the student who wrote the code in Figure 7.5 was asked to explain the flow of insertion, it took a very long time. It took him about two minutes of drawing to explain the insertion before he noticed something was wrong. This shows signs that the student has trouble with visualizing what happens in the program and how it relates out to the data structure.

One students also made this error during the observations. As he was working on the code he spent about five minutes writing the code in Figure 7.5. He spent a lot of time changing small pieces of code, starting out with a method that was about thrice as long before he shortened to this and believed it worked properly. When asked about the piece of code with the wrong sequence of pointer update he noticed immediately. For that reason we do not believe that pointers in themselves were a problem. But we believe the student spends so a lot of focus trying to keep track of the data structure that other issues arise.

Error	Category	Further analysis?
A method expands existing storage array but does not update the original array-pointer	Logic error	This error is reviewed in section 7.4.1

Table 7.3: Errors and flaws we identified considering erroneous use of arrays

7.4 Array-errors

These are errors that arise from bad use of arrays. This means errors where technical aspects of the array is utilized in an erroneous way causing the result to be wrong or efficiency to be less than ideal. The only direct error of this type we experienced and its classification is shown in Table 7.3.

Errors where an array is chosen as data structure instead of a more suiting one, would classify as a poor choice of data structure and be listed in section 7.6.

7.4.1 A method expands existing storage array but does not update the original array-pointer

This issue can in many cases be a simple case of forgetting to write a line of code. Yet issues with arrays and their nature seem to be a problem for several students.

The problem is that any changes made to the new array is lost as the previous array is never changed in a permanent reference. If we take a look at Figure 7.6 on the next page, we can see the insert-method in a class where an array stores values and the interface allows users to choose the index an element is inserted on. The method handles indexes being larger than the array length by expanding the array by 10. That causes another error where the index may be larger than the previous length + 10. But we do not look at that error here.

Encountered misunderstandings

There is one really big issue that we encountered in this error and in the explanation of arrays found with another student as well. It is the only one we found in this study.

Array-pointers are somehow different from regular pointers

This is a belief we have come through due to the explanations from multiple students as they try to explain what an array is. In the case in Figure 7.6 the author expressed an uncertainty about the method without being asked specifically about it.

```
public boolean insert(T object, int index)
{
    int newLength;
    T[] newContainer;

    if (index < container.length && index >= 0){
        if( conatiner[index] == null){
            container[index] = object;
            nrOfElements++;
            return true;
        }
    }
    else if (index > container.length){
        newContainer = (T [ ] ) new Object [index+10];
        for (int i=0; i<conatiner.length; i++){
            newContainer[i] = container[i];
        }
        newContainer[index] = object;
        nrOfElements++;
        return true;
    }
    return false;
}
```

Figure 7.6: A method that attempts to expand an array to avoid going out of bounds with a high index but fails to update the pointer to the newly generated array.
(*From the code reviews for the initial interviews*)

" We don't really know if it [the object containing the array] will keep the new container as it has a different name. "
(participant, initial interviews)

We asked the student further about the issue and if he could update the array-pointer to point to the new array. He answered:

" I'm not sure as it does not have the same length "
(participant, initial interviews)

This make us believe the student has failed to understand that an array simply is a pointer pointing to an array of pointers of the specified type including some additional fields such as `length` . Fro this understanding moving the pointer would naturally not cause any issue.

An interview with a student who did not have this error at all confirmed the belief that this misunderstanding does exist and that it causes problems with arrays. The student was attempting to debug his program and had some trouble doing that:

" I tried to print the array. But it did not work. I only got a weird ID from it. "
(participant, second round of interviews)

Obviously the student attempted to print the contents of the array by simply printing the pointer. But that is only a memory address, like any other pointer in Java. So we asked him to explain what an array truly is and he answered:

" I can't give you a concrete answer of what the concept is. As far as I understand it is a collection of variables "
(participant, second round of interviews)

Error	Category	Further analysis?
A class uses an Object-array for storing generic objects	Design error	We consider this error to be *less relevant* to the topics in the targeted course.
A method compares generic objects using the result from `toString`	Logic error	This error is reviewed in section 7.5.1
Implements generic interface without specifying class parameter sent to the interface	Logic error	This error had *too few occurrences* and is not considered common.
A private inner class is declared as generic without any reuse	Design error	This error is reviewed in section 7.5.2
A Node-class in a generic container contains a String id to compare elements	Logic error	This error had *too few occurrences* and is not considered common.

Table 7.4: Errors and flaws we identified that considers generics and class parameters

7.5 Errors in generic classes

These are errors that are caused by poor use of or inability to set up generic classes and objects. Before we take a look at the errors we experienced we will define what a generic class is.

The textbook of the targeted course defines a generic class as a class with one or more type parameters[32]. This allows a single class definition to be used on different types of objects that would otherwise need its own implementation of the class. The `ArrayList` -class mentioned previously in this chapter is an example of a generic class and it can hold any object.

But you cannot substitute the generic parameter with any of the eight primitive types in Java. Only classes are allowed.

The way we program these classes is by defining a generic incoming parameter in the class definition such as `class Container<T>` where `T` is the class we expect to use. Since we do not tell anything more about `T` we cannot say anything more about it's properties except the fact that it must extend the class `Object` , like any other Java class.

The errors we experienced considering exceptions can be found in Table 7.4.

```
public E getFromString(String key){
        Node value = null;
        boolean found = false;
        int res;
        String string;
        Node temp = first;
        while (temp != null & !found){
                string = temp.data.toString();
                res = string.compareTo(key);
                if(res == 0){
                        found = true;
                        if (debug){
                        return temp.data;
                }
                temp = temp.next;
        }
        return value;
}
```

Figure 7.7: A method that compares a generic object to a string using `toString`.
(*From the code reviews for the initial interviews*)

7.5.1 A method compares generic objects using the result from toString

This error is made by some of the the students whose assignments were reviewed for the initial interviews. The issue with doing this is that we have no way of guaranteeing the `toString` -method has any proper implementation for the classes that are used as a generic class parameter. Unlike using methods that are bound using interfaces where the author of the class is obliged to make sure it does the toString-method from the `Object` class may never be overwritten.

The method in Figure 7.7 is intended to lookup an object using a `String` -object. But the student who wrote the code has not exploited the fact that the generic class `E` is bound to an interface called `Lik` that has a method, `boolan samme(String s)` that returns true if the object can be identified using that particular `String` -object. We see the very same error in the method `find` in Figure 7.8.

Encountered misunderstandings

We encountered two different misunderstandings considering this error. They are closely related and sometimes students have them both.

toString is our only alternative to "know" anything about generic objects

This misunderstanding comes from a lack of understanding of what you get from binding a generic class parameter to an interface, such as `Comparable`. The students do not really understand that this guarantees that every class used as a parameter must have that method implemented.

The student who wrote the code in Figure 7.7 explain his choice by saying it was the only option:

> *" But I don't have any other option when I need to compare the object to a `String`. "*
> (participant, initial interviews)

In this case it may have been a bit misgiving that the method was given a `String`-object as parameter. This makes it easier to immediately get pushed into believing `toString()` is the right choice. The student even states he had problems when using different classes than `String` for testing, but did not understand why that was:

> *" I have a problem using other objects than strings as the `toString`-method does not give a good result. "*
> (participant, initial interviews)

This shows the student knows that this is not a good solution and `toString` don't really provide a good answer to the problem. But he did not manage to come up with an idea to use one of the interfaces available. We believe this shows a clear lack of knowledge of how binding generic objects to interfaces is a way to "know" more about how to use them.

Using toString to compare generic objects is not a problem

The second misunderstanding we encountered is the belief that using `toString` for this type of matter is no problem at all.

One of the students had attempted to sort the elements in a linked list using `toString().compareTo()` and never bound the generic parameter to `Comparable`, in way an expert programmer would when needing to sort generic objects.

He had not notices any problems even after testing his program:

> *" I tested this using String-objects. It worked without a hitch. "*
> (participant, initial interviews)

When asked if it would cause an issue for other classes he came up with what he believed was the solution. That was to simply implement a working `toString`-method that identified the other classes properly. His statements make us believe that some students do not see that we cannot know that the `toString`-method will be overwritten and that other classes may need the same treatment in the future.

Program robustness is not as important as other program attributes

Program robustness is mentioned for multiple errors in this chapter. The lack of consideration of how this error affects any expansions on the program lead us to believe that this errors is partially caused by the same misunderstanding that has been listed in section 7.2.1.

7.5.2 A private inner class is declared as generic without any reuse

This is an error we typically have seen when students create `Node` -classes to use inside a generic container. The issue with doing this is that the programmer is forced to explicitly state what class is used as a parameter to the `Node` -class each time a data field is defined or an instance is created. The scope of the generic parameter from the outer class does include the inner class, but redefining it overwrites the original definition. This forces a lot of extra work and makes the code a lot more messy.

In Figure 7.8 on the facing page we see a class where the author has defined both a node class and an iterator as generic. But neither would have needed to be. We also see that the student does not specify the parameter used and expects it to still be inside the scope of the outer class. This includes the class `MyIterator` where the name of the generic parameter is different.

Encountered misunderstandings

We encountered two different misunderstandings leading to this type of error. The first leads to the type of case that we see in Figure 7.8, where the class parameter for the nodes are not specified and the values need casting. The other leads to a very similar version of the issue where the parameter is set properly and no casting is required. But they may overlap.

A class name cannot be overwritten

The student who wrote the code in Figure 7.8 had problems with having to cast the data from the nodes before returning it or using the `compareTo` -method. When we asked him why those problems appeared he did not know. He said he got some compiler error about not being allowed to return the object without casting.

After being asked what type of element T was in one of the nodes he used he said it was the same as was specified for the class `SortertEnkelListe`:

> *" It a generic object of the class we specified as `T` in the class. "*
> (participant, initial interviews)

Based on this we believe the student think that because it says `T` in both places the classes are somehow the same and is not overwritten by the second specification in the `Node` -class.

```
public class SortertEnkelListe<T extends Comparable<T>
       & Lik> implements AbstraktSortertEnkelListe<T>
{
        class Node<T>{
                public Node neste;
                public T data;

                public Node(T objekt){
                        data = objekt;
                }

                public T data(){
                        return data;
                }
        }
        class MyIterator<E> implements Iterator<E>{...}

        ...

        public T find(String key)
        {
        Node temp = first;

        while(temp != null){
                if(temp.data.toString().equals(key)){
                        return (T)(sjekk.data);
                }
                temp = temp.next;
        }
        return retur;
        }
        public Iterator<T> iterator(){
                return new MyIterator<>(first);
        }
}
```

Figure 7.8: A generic class that has two inner classes that are generic as well.
(*From the code reviews for the initial interviews*)

A class must have its own generic parameter to be able to use one

We believe the student who had the problem above also has this misunderstanding. We asked him what would happen if he removed the generic parameter from the `Node` -class. His response was that he was not sure. But he expected the program to no longer work as we could not use the generic parameter without specifying it.

> *" Ehhh, I think everything demands it to be. The whole class is generic and can be used by any object that is `Comparable` and `Lik.` "*
> (participant, initial interviews)

We had this misunderstanding supported by the observations where one of the students while writing the node class said he added the generic parameter so we could use it in the node.

> *" Now I'm writing the `Node` class. I add the parameter `T` so I can use it for the data pointer. "*
> (participant, observations)

But the student who did this did specify the type of node used every time. So we believe he understood it could overwrite the existing type. But he did not understand that everything inside the `Node` -class is also inside the list and that the scope of the list extends to the private class also for generic parameters.

7.6 Poor choice of data structure

These are errors and flaws that arise in two different occasions. The first is when the programmer choose the wrong preexisting Java construct to provide a basic data structure either for storage of data or for implementing an algorithm. As the students we have reviewed are not taking a course in data structures these errors mostly involve more basic data structures such as two-dimensional arrays and the class `ArrayList`. The other occasion is when the programmer makes a poor design in their own custom data structure, creating an inefficient program as a total. It is different from the functionality distribution errors discussed in section 7.7 as the issue in this section is how data is stored, not where.

Of course there may be pros and cons when choosing a data structure. One option is not always clearly best or better than another. Yet some choices are often considered worse than others in certain situations. For these errors the context they were made in is vital.

The different flaws and errors we encountered considering choice of data structure can be found in Table 7.5 on the next page.

7.6.1 An ArrayList is used instead of an array for storage when indexes and number of elements are known

This issue bounds into the fact that some problems have an upper limit of possible data elements, whether these are instances of a class or primitive types. The Java class TTArrayList is a container that implements the `List` interface and it implements the interface using an array. It supports a few additional methods such as `public E get(int index)` and `public void add(int index, E element)`. But `ArrayList` does not support an upper limit in the way an array does. Nothing stops a program from adding additional elements beyond the initial capacity even if it has been specified.

This error was very common in the programs that we reviewed. Almost half of all students we reviewed in the second round of interviews showed this very error. One of the students produced the code in Figure 7.9 on page 77 as part of his solution to the mandatory assignment reviewed in chapter 5. The student has implemented an abstract class Field to handle storage of *N* elements of the class Square. He has also added some methods including one that returns the stored squares as an array. The error is not apparent from the code itself. Therefore we will supply you with the problem that the code is supposed to solve. The intention is to create a container to hold a fixed number of `Square` -objects, playing the role of row, column and box in a game of Sudoku. We can see that the structure used in Figure 7.9 in no way limits the amount of elements that can be inserted. Nor did any of the classes extending this abstract class.

An implementation of the `addSquare` -method can be found in Figure 7.10 on page 77.

Error	Category	Further analysis?
An ArrayList is used instead of an array for storage when indexes and number of elements are known	Design error	This error is reviewed in section 7.6.1
An ArrayList is used instead of an array when maximum length is known but final size is uncertain	Design flaw	We consider this error to be *less relevant* to the topics in the targeted course.
An object use "tags" to define which other objects it "belongs" to instead of being a pointed to by a field in the other objects	Design error	This error is reviewed in section 7.6.2
The program contains extra data structures to describe the state of the program state instead of using appropriate methods to compute the state	Design error	This error yielded mainly misunderstandings that are *covered in the analysis of other errors*.
Use of two-dimensional arrays when position is irrelevant	Design flaw	We consider this error to be *less relevant* to the topics in the targeted course.

Table 7.5: Errors and flaws we identified that considers choice of data structures

```
public abstract class Field {
    protected ArrayList<Square> squares =
     new ArrayList<Square>();
    public abstract void addSquare(Square s);
    public boolean contains(Square square) {
        boolean found = false;

        for (Square s : squares)
            if (sqare == s) found = true;

        return found;
    }
    public Square[] getSquares() {
        return squares.toArray(
            new Square[squares.size()]);
    }
    ...
}
```

Figure 7.9: Class used to store N elements in an `ArrayList`
(*From the code reviews for the second round of interviews*)

```
public void addSquare(Square s) {
    squares.add(s);
    s.setColumn(this);
}
```

Figure 7.10: Implementation of the addSquare-method from Figure 7.9
(*From the code reviews for the second round of interviews*)

Encountered misunderstandings and issues

The students whose program contains this type of error explain that they find it quicker and easier to use than an array, which would be more appropriate. Ending up with a shorter code base is mentioned by most as something positive they achieve using the `ArrayList`.

The students do not appear to be able to clearly distinct the `ArrayList` from the array. They see the similarities such as the fact that elements can be accessed using an index. The fact that a container class is used as the sole element inside another container class do not seem to be an issue to the students. This lead us to take a look at the following misunderstandings.

An upper bound of elements in a container serves no function

In all the programs we reviewed none of the students used the alternative constructor `ArrayList(int size)`. They do not specify a size for the list. We believe this reduces the chance that the students actually believe `ArrayList` supports an upper bound like the array does. We managed to find further evidence of that during the interviews. We asked the students to explain what would happen if we were to add an extra element to the `ArrayList`. Some got a bit confused by the question but all of them ended up saying it would simply add that extra element.

Since the students knew there was no upper bound it seems their misunderstanding is likely to be regarding whether an upper bound serves a function or not. We therefore took a look at what the students believed the effect the lack of an upper bound has on their programs. Only one of the students actually reasoned that his program might put too many elements in the `ArrayList` if there was a breach in his logic.

The others needed to be asked specifically if a logical error could be hidden by this lack of an upper bound. About half the students asked this question responded with a certain amount of understanding but stating that it was not something they had ever considered. The rest stated that their program was indeed correct and contained no such logic breach. Based on those statements we believe the students do not understand the purpose of an upper bound and how it helps program robustness.

Using arrays is too difficult

An additional issue we found is that a few students also showed signs of having trouble dealing with arrays and indexes. One task that proved difficult was finding the correct spot to insert elements.

In Figure 7.10 on the preceding page we can see the implementation of the method `addSquare` in of the subclasses of the abstract class in Figure 7.9. In this implementation the student does not even use the specific insert-method in `ArrayList` that uses an index. This allows him to avoid considering the position of the `Square` -object in the container. In many ways that is quite handy. In this task you do not have a strict need to know the internal position of a `Square` within the containers. If you use an array

you are forced to at least find an empty place in the array when inserting an element. That may be the very thing many students wish to avoid and find too difficult.

Multiple students state having to deal with indexes being a bother. One of the students stated he finds `ArrayList` a lot easier and prefer using it to an array.

> *" I think `ArrayList` is easier to use. I don't have to think about that kind of thing [indexes] then. "*
> (participant, second round of interviews)

Multiple students give similar statements where they say having to deal with the index can be quite bothersome.

This is supported by statements we got when taking a look at a similar issue: the almost identical case where you only know the max number of elements you need. We have classified it as a design flaw instead of an error. For that reason we have chosen to omit it from further scrutiny. But some of the data collected may be interesting none the less. Among the students one said the following when asked if he could solve a task with those assumptions using an array:

> *" That would have been problematic since I don't know the size in advance. I could have made an array with a lot of zeros. But no matter what I get a problem to decide where a value is supposed to go. "*
> (participant, second round of interviews)

In total it seems students choose to use an `ArrayList` for its simplicity. Handling arrays require a bit more work and longer code due to having to explicitly handle which position to insert an element, that many other data structures don't have. It may be that they do not realize that when they create their own container they are free to add helping structures such as a `size` -field to help with this kind of matter.

The premises for a task are not absolute

One of the students have both a comment in his code and a statement in his interview that made us realize that he did not consider the premises given in the task are absolute. The code can be found in Figure 7.11 on the next page. During an interview with the student his explanation to his choices was:

> *" I planned to support the possibility of the file containing values that don't give a fixed size. If that is possible then an array won't work as I won't know the size anymore. "*
> (participant, second round of interviews)

In this case he takes into account that the rules may not apply in the future. And if the rules don't apply then there may not be any max limit

```
// ...
// in case I decide to make some kind of
// freakshow sudoku, I'm gonna be precise.
// ...

Column[] cols = new Column[cells[0].length];
Row[] rows = new Row[cells.length];
// as I said freakshow sudoku might happen
ArrayList<Box> boxes = new ArrayList<Box>();
```

Figure 7.11: Comments indicating the task premises are ignored.
(*From the code reviews for the second round of interviews*)

to the number of squares in a dimension. For that reason he chose to use a data structure without an upper bound.

We believe this could indicate that some students may not realize the need to capitalize on the restrictions on the problems they are to solve. This shows a distinction between some types of students.

7.6.2 An object use "tags" to define which other objects it "belongs" to instead of being pointed to by a field in the other objects

This error was quite common in the solutions to one of the assignments we reviewed. About one fourth of all the subjects of our study had delivered a solution like this. In addition an equal number said they had started out with this type of solution and fixed it when either a teaching assistant or fellow student made them aware of its weaknesses.

This matter is an issue of poor design of a data structure. The result is poor performance and increased time complexity of the algorithms which uses the structure. When you look for specific objects you need to search through all the objects in the larger structure instead of pointing to them from a container. An additional issue appears if another problem causes the tags to not be unique. That would break this entire algorithm.

In Figure 7.12 on the facing page we see how a student designs a class supposed to model a column containing squares in a game board. In addition the squares have a pointer to the `Column` -object.

In Figure 7.13 on the next page the same student has created a method to aid him in finding the squares that are located in the same column. But instead of getting them directly from the object representing the column he is forced to go via the entire game board and siphon out those squares that point to the same column.

This kind of solution is also a case of an error concerning distribution of functionality in object-oriented programming that we take a look at in section 7.7.

```
public class Column{
    private static int nrOfColumns;
    private int columnNr;

    public Column(){
        this.columnNr = ++nrOfColumns;
    }

    public String toString(){
        return "" + columnNr;
    }

    public int hentAntallKolonner(){
        return nrOfColumns;
    }
}
```

Figure 7.12: This class uses an int to say which column it represents and then you search for objects that point to a column with that number like in Figure 7.12.
(*From the code reviews for the second round of interviews*)

```
public Square[] lookUpSquaresByColumn(String columnNr){
    Square[] squares = new Square[n];
    int counter = 0;

    for (Square[] row : squareArray)
        for (Square s : row)
            if (s.getColumn().equals(columnNr))
            squares[counter++] = s;

    return squares;
}
```

Figure 7.13: This code is used to search a major data structure for tags instead of pointing to elements directly from the class shown in Figure 7.12.
(*From the code reviews for the second round of interviews*)

Encountered misunderstandings

Most students who had produced this error knew that their solution was not the best. They described their own solution as kind of messy and not computationally efficient. They explained it by saying it was the solution they first tried to make work. And their reasons for not improving upon it was either lack of time or that their program already produced the correct result.

Separating functionality from data storage is good practice

The student who wrote the code in Figure 7.12 and Figure 7.13 explained his choice by calling the `Column` -class stupid and that other classes were assigned the responsible for coordinating how the program interpreted the data in it.

> *" Boxes, columns and rows are 'stupid'. They only know what number they have. They don't know anything about which squares belong to them. That allows me to use them [the numbers] later in the solving algorithm. "*
> (participant, second round of interviews)

Refactoring is not important when the program works

We believe students producing this error share at least one misunderstanding regarding the program result and refactoring. It seems they view the program as finished as soon as the result is correct. A typical answer from the students that had this kind of solution were:

> *" My program was working, so I did not change it. "*
> (participant, second round of interviews)

.

When asked about the how much refactoring he usually does one student answered:

> *" Not much really. I leave it if something works properly. It sucks if I ruin it again. "*
> (participant, second round of interviews)

This makes us believe that some students find refactoring a bit difficult and that they do not trust their own ability to improve their program.

Error	Category	Further analysis?
Almost identical classes are defined without using heritage from a super class	Design error	This error is reviewed in section 7.7.1
The program contains improper use of private internal classes	Design error	This error had *too few occurrences* and is not considered common.
Functionality is located in another class than the one it should be in	Design flaw	This error is reviewed in section 7.7.2

Table 7.6: Errors and flaws we identified that considers distribution of functionality and data between classes in a program

7.7 Data and functionality distribution errors

In section 7.6 we considered errors in how data is stored. In this section we take a look at errors in where data is stored and how functionality is distributed, on top of that data, through choice of classes and how they are designed. These kinds of errors and flaws are often noticeable as either programs with very few very large classes supporting a large amount of functionality, or a very skewed and unnatural distribution of methods between a larger number of classes.

These errors and flaws all relate quite closely to the basic principles that guide the object-oriented programmer shown in section 2.5.

The different flaws and errors we encountered considering choice of data structure can be found in Table 7.6.

7.7.1 Almost identical classes are defined without using heritage from a super class

This error was quite common in the second assignment we reviewed. The issue is one considering bloated code and difficulty editing the code. It is better to only need to add, remove or rewrite code in one place than having to do it in three places to fix the same problem. We see an example of this in Figure 7.14 on the next page. Both classes are part of the same program.

Encountered misunderstandings and issues

After talking to the students who had these errors in their assignments we came to two possible problems they had. The majority of the students list creating a super class as a possible improvement before ever being prompted to do so. The concept of heritage in itself seem to be well understood.

```
class Row{
    private int length;
    private ArrayList<Square> arr =
     new ArrayList<Square>();


    public void addSquare(Square s){
        arr.add(s);
        s.setRow(this);
    }
    ...
}

class Colmun{
    private int length;
    private ArrayList<Square> arr =
     new ArrayList<Square>();


    public void addSquare(Square s){
        arr.add(s);
        s.setColumn(this);
    }
    ...
}
```

Figure 7.14: Two almost identical classes with no heritage used. They also include methods for printing and to check if an element is present in the container.
(*From the code reviews for the second round of interviews*)

The problem at hand is not fully understood

When they describe the process of writing the different classes they did not see the similarity between them until the program was finished. The students do not appear to have understood the problem they were to solve sufficiently to make such a decision before they had a working program. Therefore we believe quite a few students have trouble reading and understanding their assignments. This is supported by our interviews with multiple teaching assistants in the targeted course during our preparation to review the assignments.

Refactoring is not important when the program works

We prompted the students to explain why they did not change their code when they realized that it was an improvement. Similarly to the question of refactoring a poor solution in section 7.6.2 many students said it was not important as their program already produced the correct result. Some also listed time limitations as a factor. So it might be that they would have refactored the code given time.

For this reason we believe the conclusion on the same misunderstanding as in section 7.6.2 applies here as well. Some students do not understand the importance of refactoring or have problems doing it, fearing they will cause more harm than good by trying.

7.7.2 Functionality is located in another class than the one it should be in

This is the most common error that programs we reviewed had. It was apparent in all programs but a few. Some to a larger extent than others. This issue is overlapping with some of the other issues. So we did not always address this issue directly. Our impressions are therefore partially based on things we learned from addressing other errors and flaws. The code in Figure 7.12 on page 81 is a good example of this as it contains this error as well.

The problem with programs that has poor code placement is that the code gets really hard to read, understand and change. In many cases the functionality is affected so that scaling the program or reusing components becomes almost impossible. The object-oriented principles are explained in section 2.5.

We can see this error well in Figure 7.15 on the next page. Instead of implementing a `contains` -method which most containers should have, this implementation forces anyone who uses the container to see for themselves. If the implementation of `Row` was to change many other parts of the program would need so as well.

```
public class Row {
    Square[] row;

    public Rad(int size) {
        this.row = new Square[size];
    }
    public void insert(Square s) {...}
    public void remove(Square s) {...}
}
...
public boolean isThisUsed(int a) {
    boolean used = false;
    ...
    for (int i = 0; i < row.row.length; i++) {
        int tall = Integer.parseInt(
            row.row[i].toString());
        if (tall == a) {
            used = true;
        }
    }
    ...
    return used;
}
```

Figure 7.15: This class contains a lot of data but no functionality to check if it contains an object is in a method inside a different class body.
(*From the code reviews for the second round of interviews*)

Encountered misunderstandings

MVC is fully compatible with object-oriented principles

This case is quite special. One student had used a completely different set of principles when designing his program than in any other program we reviewed. The student had tried to model his program after the Model-View-Controller principles. A set of principles that places all program functionality within the Controller, the storage description within the Model and user interface within the View. These principles are designed for dealing with applications with advanced user interfaces. MVC is not well suited for solving the kind of tasks given in the targeted course as well as being in conflict with several object oriented principles.

> *" [The class]* `Sudoku` *is the controller,* `Board` *is the model og* `GUI` *is the view. The board was to know everything, squares, columns, boxes and rows. "*
> (participant, second round of interviews)

Especially the use of a single class as the Controller made the decomposition of the program quite poor. Most container classes ended up having no functionality at all, while the class meant to represent the Controller had way too much functionality included. In Figure 7.12 on page 81 we see how some classes become very empty while Figure 7.13 on page 81 is an example of functionality that belong within Figure 7.12 according to object-oriented principles, yet is present in the class representing the Controller.

When asked about his choice the student answered that it seemed natural as the assignment was supposed to have a graphical user interface. A follow-up question revealed that he thought object-orientation and MVC was about two completely different things. The first about the abilities of a language and the second about how to design your program:

> *" Isn't object-orientation about objects and stuff? While MVC is how you put it all together, function and display? "*
> (participant, second round of interviews)

Distribution of functionality does not matter

When asked about their choices of method location many of the students did not really grasp that there are other concerns to code quality than code length. They were not all aware that certain functions naturally belong in certain classes. Some of the students simply considered certain classes as a set of pointers with no true functionality or methods contained within them. This naturally created a heavy weight on the classes referencing those classes.

We asked the students with bad distribution of functionality specifically about how they chose to design these things. One of the students gave an answer that sums what we heard from almost all of them:

> *" I don't really know. I put it where it feels natural. If I see that I have similar code in two locations I see if it can be put together as one method. "*
> (participant, second round of interviews)

We believe the students are aware that moving code may improve upon a program. But it seems they limit improvements to those that reduce the amount of code. This excludes improvements made to the way semantics of a class is designed to achieve a "natural" role in the program. We see that they do not include the design-aspect of object-orientated programming explicitly into their arguments. None of the terms usually used in object-oriented design were mentioned when they explained their choices.

We believe this shows a lack of understanding that object-orientation also plays a strict role in design. They may have some implicit understanding of many of the concepts used. But why they matter is not something they understand beyond the obvious improvements in size and time spent.

From the statements of the student using MVC we believe he the did not understand that object-oriented programming is not simply a tool for writing code. He chose to fill the role usually filled by object-oriented design with the MVC-paradigm.

Chapter 8

Discussion

In this chapter we discuss the misunderstandings we identified during the interviews and observations. In chapter 7 we focused on the errors and how we found the misunderstandings. In this chapter we take a look at possible sources of these misunderstandings and compare our findings with those made by other researchers. Finally we discuss if and how our findings would impact an attempt at using pre-defined misunderstandings to improve summative and formative assessment in first year programming courses.

We separate the misunderstandings into two main groups as we discuss how they relate to each other, their origin and extent. We also take a look at what effects they may have overall. In the final section of this chapter we take a look at how the misunderstandings we identified may be utilized in assessment methods in programming courses.

One of the misunderstandings we identified, *"IDE solutions to problems are good enough"*, does not fit into the two groups we chose. It would need a group of its own for misunderstandings concerning tools aiding us in writing code. However, this type of misunderstanding is not very relevant to this study. The targeted course do not teach the use of tools such as IDEs and the students' skills with them is not target for assessment.

8.1 Misunderstandings concerning technical aspects of programming

The second type of misunderstandings we take a look at are misunderstandings concerning skills and knowledge required to write Java code at the level of the targeted course. This include knowledge of some data structures, Java and programming languages in general. This means misunderstandings of specific classes, components or structures that are frequently used or provided by the programming language. The technical misunderstandings we identified in chapter 7 are displayed in Table 8.1 on the following page.

We see from this that all of the technical misunderstandings we identified are about either exceptions, arrays, generics or data structures

Misunderstanding	Theme
Exception type is not important when handling exceptions	Exceptions
Exceptions do not adhere to the same rules a other Java-classes	Exceptions
Array-pointers are somehow different from regular pointers	Arrays
Using arrays is too difficult	Arrays
toString is our only alternative to "know" anything about generic objects	Generics
Using toString to compare generic objects is not a problem	Generics
A class must have its own generic parameter to be able to use one	Generics
A class name cannot be overwritten	Generics
Data structure visualization is too difficult	Data structures

Table 8.1: Misunderstandings we identified regarding technical aspects of Java and programming languages in general

in general. All of these these misunderstandings regard topics that are expected knowledge from students of the targeted course.

The misunderstandings we identified were both from errors that cause an erroneous program result and errors that are related to design. Some of the misunderstandings have caused both. We believe this shows that a working program does not always prove that the student have a working understanding that would always lead to a working program. Since they ended up with a working program in the assignment we reviewed we could argue that they do have the proper understanding to attain a working program later. This leads to an argument of what proper understanding is. Lauvås'[37] argument that many students deal with summative assessment using bad learning techniques does apply in this case.

Bad learning techniques can be used to attain a good result on an assignment without resulting in a proper understanding of the concepts involved. An example of this is that we do not know exactly how the working program was created. Some of the details were unfortunately lost before we conducted our interviews. It is not unlikely to assume it could have been made with help from sources that provide code examples or using a trial and error approach. A good way to understand this is to look at how novices debug programs to fix their errors. Murphy et al.[42] found that a lot of novices apply ineffective techniques for debugging their programs, techniques that show little understanding of the problem they are solving. The debugging often lacks aim and keeps on until the novice gives up or reaches the goal by accident.

8.1.1 The importance of misunderstandings concerning arrays

It is quite interesting that there are multiple misunderstandings considering arrays among the misunderstandings we identified . Arrays are part of the topics that the targeted course does not really teach at all. It is expected that all students of the course have had proper introduction to arrays from previous courses. But it seems arrays are not as well understood as the teaching staff of the targeted course expect. We did not experience a huge amount of errors that considered arrays directly. But we did learn that many students choose other data structures provided by Java. They find arrays difficult to handle and prefer using classes that give a simpler set of methods to deal with when inserting and looking for elements. Our findings may indicate that problems using arrays are more widespread than previously expected.

The misunderstandings we identified concerning arrays were not only misconceptions where the students have developed a consistent understanding that is wrong. They seem to have trouble explaining the very concept and are in no way certain in their choices or knowledge. That indicates that they may not have gotten enough practice with the concept.

The choice to use classes from the Java library over arrays may indicate that the use of arrays often cause an inconvenient number of bugs. These are bugs are time consuming for the students. This would be in line with a study made by Soloway, Bonar, and Ehrlich[49]. Their study shows that novice programmers more often use strategies for looping through data that are prone to contain bugs such as the "off-by-1" bug where reading and processing data is not aligned properly. If the students are unable to properly set up looping strategies, that may explain why they prefer using classes that provide simple methods for inserting and looking up data.

8.1.2 Other technical misunderstandings

The other technical misunderstandings are all about topics that the targeted course attempts to teach its students. A small discussion only about misunderstandings considering exceptions is held in section 7.2.1. Most of the misunderstandings concerning exceptions and generics share one common trait. They all seem to revolve around inheritance of fields and methods. This is not strange considering both exceptions and generic class parameters are in a way specialized classes in Java. The exception is technically just a class with an additional ability of being thrown, and a generic parameter is a placeholder for a class of a certain type just like a pointer to an interface.

It seems some of the students who took the targeted course have not properly understood how these types follow the same rules as other classes. Since they are used in a very different way than regular classes they may never have seen this similarity. We do not really know if the other students in the targeted course see this either. We know they manage to write their programs correctly. But that can just as easily be attained through learning the "patterns" of the syntax or copying a working example

from the lectures or somewhere else on the Internet.

We did experience one of these uses of copying and pasting code in one of the observations we conducted. The subject was tasked to write a LIFO-queue using linked list structure. As he was not sure how to write a linked list he quickly decided to try to find an appropriate implementation on the Internet or among previously completed assignments. He ended up using code from a sorted linked list, which has completely different requirements than a LIFO-queue. The methods in the Java class were barely changed before the subject considered them finished and submitted the code. This shows how many students are rather careless when copying code, often not understanding the code in which they copy. By using such a method with trial and error it is possible to attain a working program without ever understanding the basics of it.

It is interesting to note that there is a very limited amount of existing research in Computer Science Education concerning generic classes. This provides us with little data to compare our findings. This may be explained by generic classes not being among the original features of Java. It was introduced with the Java Developer Kit (JDK) 5.0 in 2004 but took some time getting traction in courses in object-oriented programming such as the targeted course. Yet courses have taught the concept for at least half a decade and little research has been produced.

8.2 Misunderstandings concerning object-oriented practice and the importance of program design

The last group of misunderstandings are those concerning object-oriented practice and the importance of program design. These are misunderstandings concerning *how* to use the technical constructs provided to us by the Java programming language in a good way.

Even though this class of misunderstanding contains fewer categories than the technical misunderstandings discussed in the section above, this type of misunderstanding was far more frequent among the students we interviewed or observed. All but very few of our subjects had at least one of these misunderstandings.

Matching these misunderstandings with the errors they were the cause of in chapter 7 we see they cause errors in both program result and program design. Most of the misunderstandings, however, cause only design errors and design flaws. This is not surprising as they mostly relate to good practice and priority of other aspects of a program than which result it produces. But some misunderstandings, such as the lack of focus on program robustness, may cause errors that have an effect on the results of a program.

Misunderstanding	Java theme
Program robustness is not as important as other program attributes	Program design
An upper bound of elements in a container serves no function	Data structures
The premises for a task are not absolute	Program design
Separating functionality from data storage is good practice	Program design & object-oriented practice
Distribution of functionality does not matter	Program design & object-oriented practice
MVC is fully compatible with object-oriented principles	Program design & object-oriented practice
Refactoring is not important when the program works	Program design
The problem at hand is not fully understood	Program design

Table 8.2: Misunderstandings we identified regarding object-oriented practice and the importance of program design

8.2.1 Misunderstandings causing errors in program result or compilation

The few misunderstandings of this category that may have lead to errors in result or compilation are mostly about program robustness. Our experience shows that misunderstandings that do not relate directly to the technical aspects of a solution still may lead to technical errors. This is very interesting indeed. Students' erroneous notions of what is important in a program may lead to the very thing they find important being compromised.

But there were not that many different misunderstandings that we linked directly in such a way. Our findings may be wrong in giving it this direct link. The misunderstandings are still very likely to be present for many students. But as we identified multiple possible misunderstandings that cause those errors, this shared misunderstanding, may simply occur from chance.

8.2.2 Misunderstandings causing design errors and flaws

The many occurrences of these misunderstandings in this study may have a natural explanation. The assignments we reviewed spanned up to three weeks. For most students that should be enough time to get rid of most errors in result or compilation. The students of the targeted course are taught to test their programs so we may expect them to, more often than

not, be able to see if their result is correct. This leaves a lot of errors regarding how they achieve this result.

We have seen two distinct ways these misunderstandings turn in our results. The first way is a lack of understanding the factors that are important to write a good a program, such as scalability, efficiency and robustness. Ragonis and Ben-Ari[45] identify multiple misunderstandings that cause issues in design, such as misconceptions about modularity and information hiding. Interestingly Ragonis and Ben-Ari[45] also identified a lot of technical misunderstandings that we did not experience in this study. The subjects in their study are students that have no prior experience with programming, while our subjects have at least one semester. Yet they share a lot of misunderstandings about how to design a program.

The second way is a misunderstanding considering whether such aspects of a program are important in an assignment solution. In a study to identify misunderstandings about object-oriented design, using code reviews by novice programmers, Turner et al.[53] noticed one very interesting point. While they found lack of knowledge and understanding if several concepts used in OOD, they also summarize one particular trait shared by the novices' reviews:

> "... indicating, what is common knowledge to instructors, that students tend to write code to meet the assignment specification to the letter and no more."
> (Turner et al., p. 100, [53])

This may apply to what we have experienced in this study. The students often seemed happy as soon as the specifications given in the assignment text appeared to be fulfilled. Their choices when it comes to refactoring and sometimes quite good knowledge about points that may have been improved does support this theory.

Some may argue that these types misunderstandings are the same. The students who stick to bad solutions because they don't bother to change them don't understand the factors that make a program good. But multiple students in our study were able to point out clear weaknesses in their programs, sometimes even without being asked about them. That shows that the students are aware that their solution do not fit the criteria to be "good". In other cases the students are not aware their programs has weaknesses and why they matter. This shows a clear boundary between the two types.

Misunderstandings about object-oriented design (OOD)

When reading the course title we expect the students to at least know of the principles used when applying object-oriented programming (OOP) to solve a problem. They are the reasons why we use the concepts encapsulation, composition, inheritance and delegation we explained back in section 2.5. If we take a look at the learning objectives stated on the course website[16] we do not find any mentioning of the principles of

object-oriented programming. That may explain why a lot of the students seem to not understand these principles.

This coincides with the results of Turner et al.[53]. They experienced that many novices have problems understanding the intention behind the use of both decomposition and encapsulation. The novices tend to see proper use of encapsulation as overkill, misunderstanding encapsulation with abstraction. Some of them seem to prefer long blocks of code, explaining that they are easier to maintain than multiple smaller classes. We experienced both of them in how some of the subjects in the observations designed their classes in the first task. They attempt to keep the amount of classes and interfaces low by sacrificing proper encapsulation of data. Ragonis and Ben-Ari[45] also identified a lot of misconceptions considering encapsulation among novice programmers.

Eckerdal et al.[17] experience that students, even those who are graduating from university, do not know how to properly design programs. More surprisingly, their research show that the ability to design software do not correlate to academic performance. That may indicate that we were wrong about a lot of the "more skilled" students which we never invited to interviews because they did not have enough errors or flaws in their programs. They may either have flaws in their programs that we did not notice or they may not be "more skilled" than the ones we interviewed as far as academic results show.

But even though their actions may match we do not experience them to be misconceptions about the principles of OOD and how they should apply the concepts. Except from the student who attempted to solve his assignment using the MVC-principles none of the students made many arguments for the their choices using design principles. Their statements are similar to some of the goals expressed in the American approach to OOD that was introduced in section 2.5.1, such as shorter code. That may be caused by a "selfish" wish to make their own work as easy as possible. It does not have to be caused by them knowing only the American approach to OOD.

Unfortunately this was not something we focused enough on in our questions. Therefore we do not really have any obvious data showing us the extent of the students' knowledge of the or the concepts of OOP or the OOD principles applied when arguing for the use of the concepts. But we do have the feeling that the students do have of any structured reasoning behind their design choices at all. This seems likely if we take into account that the course does not teach the concepts or any of the approaches explicitly.

The importance of OOD-misunderstandings

The OOD-misunderstandings almost always show up in relation to other misunderstandings causing error in the students' programs. The different misunderstandings may not be be related at all. But we believe that the OOD-misunderstandings may cause a lot of other misunderstandings related to good practice and good design. The secondary misunderstand-

ings then cause even more errors. If that is true then remedying the OOD-misunderstandings may lead to a reduction in other misunderstandings and improvement of issues that are not directly caused by misunderstandings of OOD.

Such a claim would need further verification. A study must be designed to include students with a better mix of skill and academic results and fewer other errors, as well as getting a better look at how the code made in the solutions of an assignment develops over a period of time. A case study could be suitable.

8.3 Assessing the misunderstandings

One of the questions we wanted to answer in this study is: Is it feasible to use these misunderstandings as an aid in assessing students? We will take two different looks at it. The first is basing an assessment method on misunderstandings alone. Another is to use them as additional data to existing methods.

8.3.1 Using only misunderstandings

We suggest two different strategies for the examiner to assess a student based only on the misunderstandings the student has and use the misunderstandings we have identified to see if the nature of the misunderstandings support them.

- If the examiner knows the misunderstanding in depth he can make a direct assessment of how far away from the understanding stated by the learning objectives of the course the student is.
- If the examiner only knows the seriousness of the misunderstanding he can assess how he can make an educated guess towards how far away from the understanding stated by the learning objectives of the course the student is.

The first would be a viable strategy for both formative and summative assessment. The second is unlikely to be viable for formative assessment. Formative assessment is supposed to help the student improve his understanding. A metric of how far away from a good solution a student has come is unlikely to help him correct the actual misunderstandings.

From our findings in this study we can say pretty safely that few, if any, errors can only be caused by one misunderstanding. We have found multiple possible misunderstandings that seem plausible in causing an errors. It was not uncommon that students even had more than one misunderstanding considering one single error.

Earlier we discussed how simple errors sometimes seem to hide more serious misunderstandings than what is obvious. We may ask an additional question that is very relevant: Is the opposite possible as well? In the observations we experienced that some students had grave

errors caused by simple typographical errors. While the exercises in the observations do not fit the criteria for being a programming assignment, it is not unlikely to assume that such cases exist in programming assignments as well.

In this study we were able to learn which misunderstandings a student may have based on lengthy code reviews, interviews and observations. But examiners would probably not have the time and resources to do this. It would at least require most computer science courses to completely remake their assessment schemes. It does not seem likely that any method to assess an assignment can rely on judging misunderstandings alone. The work needed to identify them all would be very time consuming.

8.3.2 Supplementing with misunderstandings

Improving existing methods for assessment with the most likely misunderstandings seems like a more viable alternative. In many cases examiners most likely already try to figure out why a student made some error whether he uses an holistic or analytic approach to assessment. Although we have not been able to identify all possible misunderstandings in the targeted course, it would be possible to extend our work and get more data on which misunderstandings are most likely to have caused a specific error.

A list of errors and likely misunderstandings could be given to the examiner. The examiner would then use that list to guide him in gaining insight into the understanding of the student being assessed. If the student has multiple errors in his assignment, the examiner can compare the likely misunderstandings from the different errors and improve his chances to get his assessment right.

In formative assessment

In formative assessment it would be possible for the feedback given to a student to be formed based on the likely misunderstandings he has. Say we have a case where a student has both an error in a method that handles an array and he chooses an `ArrayList` or other container from the Java library in many places in the solution. The misunderstandings we experienced would indicate that the student has trouble understanding how to design methods to manage the data in the array. Feedback given the student can be customized to include helping points on how to design methods that treat looping structures and array structures:

> *Your solution has some weaknesses in regards to using arrays. When you choose a structure to store data in, an array is a good choice but often requires designing extra methods to deal with insertion, deletion and lookup of data. Storing the index of the next free spot and using for-each-loops to look for elements will be helpful in designing these methods. After the methods have been designed the rest of the program can access the data in the array through them and serve as its own container similarly to that of* `ArrayList`*.*
> (Example feedback)

In summative assessment

A similar approach can be used in summative assessment. Instead of designing customized feedback to the student the examiner could use the likely misunderstandings to help him make an educated guess at how much the student understands. That level of understanding can be compared to the one expected by the course, expressed through the intended learning objectives. A scoring system can be devised based on the different levels of understanding.

Chapter 9

Conclusions and future work

In this thesis our goal was to identify errors that are present in assignments submitted by students of a first year programming course and the misunderstandings that may have caused those errors. In addition we wanted to answer whether it would feasible to use the misunderstandings as a metric for assessing the students who write the assignments. In this chapter we take a look at how our results answer the questions stated in section 1.1, suggest some possible ways to address the misunderstandings we identified and present how our work can be used in future research.

9.1 Conclusions

From reviewing the assignments of 23 students of the targeted course, written in Java, we have identified multiple errors that appear to be common in submitted assignments. The assignments were "take-home" assignments, spanning up to three weeks, that every students needed to get approved to be allowed to attend the final exam. The errors we have found may vary in other types of instruments of assessment such as multiple choice tests and classroom exams.

9.1.1 Errors

> *Which programming errors do students of a first year programming course make in their assignments?*

Our findings show that almost no students deliver programs that do not compile. Syntax and semantic errors are barely present in the list of errors we experienced during our code reviews.

The assignments we reviewed considered most of the topics covered by the targeted course. Of these topics lists, exceptions and writing generic classes appear to be the most common problem areas for the students whose solutions we reviewed. Previous research into errors considering generics is lacking and our study presents multiple errors that were common among novice programmers in the targeted course.

The majority of the errors present in assignments do not cause compiler errors, runtime errors or a faulty result. Instead they are errors and flaws

that cause the program to be sub-optimal in the measurements of efficiency, scalability or robustness. We saw many students struggling to design their programs well. The most common flaw of all was code placement that was in direct conflict with one or more of the principles of object-oriented programming. These errors make code hard to read, expand upon and often coincide with other errors.

9.1.2 Misunderstandings

Which misunderstandings may have caused these errors?

From interviews or observations of a total of 23 students we have identified misunderstandings that seem to cause the errors we identified during code reviews. The misunderstandings may differ for other types of instruments of assessment than the one we analyzed.

The misunderstandings we identified include both technical misunderstandings of constructs used in Java and the art of using them. In our discussion in chapter 8 we disclose findings showing how the different types of misunderstandings relate to each other and whether they cause different or similar outcomes for different students.

9.1.3 Assessing the misunderstandings

Is it feasible to use these misunderstandings as an aid in assessing students?

- *In formative assessment?*
- *In summative assessment?*

As expected our findings show that errors may be caused by multiple misunderstandings and no one-to-one relation between errors and misunderstandings was found. We also learned that the one misunderstanding may cause errors that do not have an effect on the program result, as well as errors that do. That shows that a working program may "hide" misunderstandings that should have additional impact on the feedback given to the student who wrote it.

We use that to conclude that the examiner can never be certain in his interpretation of a student's understanding based on code alone. Studies can supply the examiner with predictions on which misunderstandings are likely or not, but not definitive answers.

This makes an approach to assessment using only misunderstandings unfeasible. Identifying the correct misunderstandings would require a lot of additional work when using the most common assessment schemes such as written exams or multiple choice tests.

Predictions of likely misunderstandings, given certain errors, may provide good additional data to better equip the examiner to effectively assess a program. In formative assessment the predictions can be used to help the examiner present helpful feedback that attempts to address the likely misunderstandings. In summative assessment the predictions can be

used to help create a better guess at the level of understanding a student has considering different topics. That level of understanding will aid in deciding the appropriate grades for the student.

9.2 Suggestions to remedy the misunderstandings

One suggestion to remedy some of the misunderstandings of the students of the targeted course would be to introduce object-oriented design principles in an explicit way. Our belief is that by making the students aware of the many additional metrics we use when we assess the quality of a program we ensure they are aware that program result and lines of code is not all they must consider.

We would also suggest that the number of mandatory assignments in the course either be reduced or the score requirement removed. We experienced how the students worked with the primary goal to be done in time for the deadline instead of focusing on writing good programs and improve their learning, similar to what existing research into continuous summative assessment has found[37].

9.3 Future work

We recommend that the work to identifying common errors and the misunderstandings that cause them be continued for a larger amount of students of the targeted course. A statistical model predicting the likelihood of a certain misunderstanding causing an error can be used to aid in assessment. This model should be tested for validity as a metric for both formative and summative assessment. For formative assessment the tests could be designed to measure how well the created feedback fits the student through interviews before and after the feedback is given to see if the misunderstandings have been addressed. For summative assessment a comparison against the results of assessment methods such as automatic assessment and holistic and analytical methods without the aid of a statistical model.

Another possibility would be to focus on gaining further insight into the nature of the misunderstandings to be able to suggest remedies for the technical misunderstandings. We would recommend doing multiple case studies of students throughout an entire assignment to experience the students process of solving a complete assignment. In such case studies it would be interesting to also review some of the students with better academic performance to see if their misunderstandings overlap with those of the students who perform worse.

Last it would be very interesting to expand the study to include INF1000, one of the prerequisites of the targeted course, to look at the level of understanding over a longer period of time. In such a study it would be possible to compare the misunderstandings of students at the end of INF1000 to those they develop during their time with the targeted course. As seen in the example of arrays some students begin the targeted

course lacking some of the knowledge they are expected to have. Do these misunderstandings cause further problems for the student in the targeted course?

Bibliography

[1] Amjad Altadmri and Neil C.C. Brown. "37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data." In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE '15. Kansas City, Missouri, USA: ACM, 2015, pp. 522–527.

[2] John Biggs. "Aligning teaching and assessment to curriculum objectives." In: *Imaginative Curriculum Project, LTSN Generic Centre* 12 (2003).

[3] John B. Biggs and Kevin F. Collis. *Evaluating the Quality of Learning: The SOLO taxonomy*. New York: Academic Press, 1982.

[4] John B. Biggs and Catherine Tang. *Teaching For Quality Learning At University*. 4th ed. SRHE and Open University Press Imprint. McGraw-Hill Education, 2011.

[5] Benjamin S. Bloom, Max D. Engelhart, Edward J. Furst, Walker H. Hill, and David R. Krathwohl. *Taxonomy of educational objectives. The classification of educational goals. Handbook 1: Cognitive domain*. New York: Longmans Green, 1956.

[6] Neil C.C. Brown and Amjad Altadmri. "Investigating Novice Programming Mistakes: Educator Beliefs vs. Student Data." In: *Proceedings of the Tenth Annual Conference on International Computing Education Research*. ICER '14. Glasgow, Scotland, United Kingdom: ACM, 2014, pp. 43–50.

[7] Neil C.C. Brown, Michael Kölling, Davin McCall, and Ian Utting. "Blackbox: A Large Scale Repository of Novice Programmers' Activity." In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. SIGCSE '14. Atlanta, Georgia, USA: ACM, 2014, pp. 223–228.

[8] Kevin Buffardi and Stephen H. Edwards. "Reconsidering Automated Feedback: A Test-Driven Approach." In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE '15. Kansas City, Missouri, USA: ACM, 2015, pp. 416–420.

[9] Janet Carter, Kirsti Ala-Mutka, Ursula Fuller, Martin Dick, John English, William Fone, and Judy Sheard. "How Shall We Assess This?" In: *Working Group Reports from ITiCSE on Innovation and Technology in*

Computer Science Education. ITiCSE-WGR '03. Thessaloniki, Greece: ACM, 2003, pp. 107–123.

[10] Charles C. Chan, M. S. Tsui, Mandy Y. C. Chan, and Joe H. Hong. "Applying the Structure of the Observed Learniing Outcomes (SOLO) Taxonomy on Students Learning Outcomes: An empirical study." In: *Assessment & Evaluation in Higher Education* 27.6 (2002), pp. 511–527.

[11] The Oracle Corporation. *Java Platform, Standard Edition 8 API Specification*. URL: http://docs.oracle.com/javase/8/docs/api/ (visited on 10/11/2015).

[12] John W Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 2013.

[13] Charlie Daly. "RoboProf and an Introductory Computer Programming Course." In: *Proceedings of the 4th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education*. ITiCSE '99. Cracow, Poland: ACM, 1999, pp. 155–158.

[14] Holger Danielsiek, Wolfgang Paul, and Jan Vahrenhold. "Detecting and Understanding Students' Misconceptions Related to Algorithms and Data Structures." In: *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*. SIGCSE '12. Raleigh, North Carolina, USA: ACM, 2012, pp. 21–26.

[15] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. "All Syntax Errors Are Not Equal." In: *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*. ITiCSE '12. Haifa, Israel: ACM, 2012, pp. 75–80.

[16] University of Oslo Department of Informatics. *INF1010 - Object oriented programming*. URL: http://www.uio.no/studier/emner/matnat/ifi/INF1010/index-eng.html (visited on 02/15/2015).

[17] Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, and Carol Zander. "Can Graduating Students Design Software Systems?" In: *SIGCSE Bull.* 38.1 (Mar. 2006), pp. 403–407.

[18] Stephen H. Edwards, Zalia Shams, and Craig Estep. "Adaptively Identifying Non-terminating Code when Testing Student Programs." In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. SIGCSE '14. Atlanta, Georgia, USA: ACM, 2014, pp. 15–20.

[19] John English. "Experience with a Computer-assisted Formal Programming Examination." In: *SIGCSE Bull.* 34.3 (June 2002), pp. 51–54.

[20] John English and Phil Siviter. "Experience with an Automatically Assessed Course." In: *SIGCSE Bull.* 32.3 (July 2000), pp. 168–171.

[21] Noel Entwistle, Abigail Entwistle, et al. “Revision and the experience of understanding.” In: *Marton, Ferrence, Hounsell, D.ai & Entwistle, Noel(Eds) The Experience of Learning: Implications for Teaching and Studying in Higher Education* 2 (1997), pp. 145–155.

[22] Nickolas Falkner, Rebecca Vivian, David Piper, and Katrina Falkner. “Increasing the Effectiveness of Automated Assessment by Increasing Marking Granularity and Feedback Units.” In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. SIGCSE ’14. Atlanta, Georgia, USA: ACM, 2014, pp. 9–14.

[23] Sally Fincher and Marian Petre. *Computer science education research*. CRC Press, 2004.

[24] Sue Fitzgerald, Brian Hanks, Raymond Lister, Renee McCauley, and Laurie Murphy. “What Are We Thinking when We Grade Programs?” In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. SIGCSE ’13. Denver, Colorado, USA: ACM, 2013, pp. 471–476.

[25] Ursula Fuller, Colin G. Johnson, Tuukka Ahoniemi, Diana Cukierman, Isidoro Hernán-Losada, Jana Jackova, Essi Lahtinen, Tracy L. Lewis, Donna McGee Thompson, Charles Riedesel, and Errol Thompson. “Developing a Computer Science-specific Learning Taxonomy.” In: *SIGCSE Bull.* 39.4 (Dec. 2007), pp. 152–170.

[26] Matheus Gaudencio, Ayla Dantas, and Dalton D.S. Guerrero. “Can Computers Compare Student Code Solutions As Well As Teachers?” In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. SIGCSE ’14. Atlanta, Georgia, USA: ACM, 2014, pp. 21–26.

[27] David Ginat and Eti Menashe. “SOLO Taxonomy for Assessing Novices’ Algorithmic Design.” In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE ’15. Kansas City, Missouri, USA: ACM, 2015, pp. 452–457.

[28] Richard Gluga, Judy Kay, Raymond Lister, Sabina Kleitman, and Tim Lever. “Over-confidence and Confusion in Using Bloom for Programming Fundamentals Assessment.” In: *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*. SIGCSE ’12. Raleigh, North Carolina, USA: ACM, 2012, pp. 147–152.

[29] Wynne Harlen and Mary James. “Assessment and Learning: differences and relationships between formative and summative assessment.” In: *Assessment in Education: Principles, Policy & Practice* 4.3 (1997), pp. 365–379.

[30] Matthew Hertz. “What Do "CS1" and "CS2" Mean?: Investigating Differences in the Early Courses.” In: *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. SIGCSE ’10. Milwaukee, Wisconsin, USA: ACM, 2010, pp. 199–203.

[31] Colin Higgins, Pavlos Symeonidis, and Athanasios Tsintsifas. "The Marking System for CourseMaster." In: *Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education*. ITiCSE '02. Aarhus, Denmark: ACM, 2002, pp. 46–50.

[32] Cay S Horstmann. *Big Java: Late Objects*. John Wiley & Sons, 2013.

[33] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. "Identifying and Correcting Java Programming Errors for Introductory Computer Science Students." In: *SIGCSE Bull.* 35.1 (Jan. 2003), pp. 153–156.

[34] James Jackson, Michael Cobb, and Curtis Carver. "Identifying Top Java Errors for Novice Programmers." In: *Frontiers in Education, 2005. FIE '05. Proceedings 35th Annual Conference*. Oct. 2005, T4C–T4C.

[35] Matthew C. Jadud. "Methods and Tools for Exploring Novice Compilation Behaviour." In: *Proceedings of the Second International Workshop on Computing Education Research*. ICER '06. Canterbury, United Kingdom: ACM, 2006, pp. 73–84.

[36] Kuba Karpierz and Steven A. Wolfman. "Misconceptions and Concept Inventory Questions for Binary Search Trees and Hash Tables." In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. SIGCSE '14. Atlanta, Georgia, USA: ACM, 2014, pp. 109–114.

[37] Per Lauvås. "Changing assessment practices in higher Norwegian education." In: *Balancing dilemmas in assessment and learning in contemporary education*. Ed. by Anton Havnes and Liz McDowell. Routledge, 2007, pp. 157–168.

[38] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. "Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy." In: *SIGCSE Bull.* 38.3 (June 2006), pp. 118–122.

[39] Ritch Macefield. "Usability studies and the Hawthorne Effect." In: *Journal of Usability Studies* 2.3 (2007), pp. 145–154.

[40] Ole Lehrmann Madsen and Birger Møller-Pedersen. "What object-oriented programming may be-and what it does not have to be." In: *ECOOP'88 European Conference on Object-Oriented Programming*. Springer. 1988, pp. 1–20.

[41] Davin McCall and Michael Kolling. "Meaningful categorisation of novice programmer errors." In: *Frontiers in Education Conference (FIE), 2014 IEEE*. IEEE. 2014, pp. 1–8.

[42] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. "Debugging: The Good, the Bad, and the Quirky – a Qualitative Analysis of Novices' Strategies." In: *SIGCSE Bull.* 40.1 (Mar. 2008), pp. 163–167.

[43] Kristen Nygaard and Ole-Johan Dahl. "History of Programming Languages I." In: ed. by Richard L. Wexelblat. New York, NY, USA: ACM, 1981. Chap. The Development of the SIMULA Languages, pp. 439–480.

[44] David M. Olson. "The Reliability of Analytic and Holistic Methods in Rating Students' Computer Programs." In: *Proceedings of the Nineteenth SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '88. Atlanta, Georgia, USA: ACM, 1988, pp. 293–298.

[45] Noa Ragonis and Mordechai Ben-Ari. "A long-term investigation of the comprehension of OOP concepts by novices." In: (2005).

[46] Rami Rashkovits and Ilana Lavy. "Students' Misconceptions of Java Exceptions." In: *Knowledge and Technologies in Innovative Information Systems*. Springer, 2012, pp. 1–21.

[47] Riku Saikkonen, Lauri Malmi, and Ari Korhonen. "Fully Automatic Assessment of Programming Exercises." In: *SIGCSE Bull.* 33.3 (June 2001), pp. 133–136.

[48] Judy Sheard, Angela Carbone, Raymond Lister, Beth Simon, Errol Thompson, and Jacqueline L. Whalley. "Going SOLO to Assess Novice Programmers." In: *SIGCSE Bull.* 40.3 (June 2008), pp. 209–213.

[49] Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich. "Cognitive Strategies and Looping Constructs: An Empirical Study." In: *Commun. ACM* 26.11 (Nov. 1983), pp. 853–860.

[50] Christopher W. Starr, Bill Manaris, and RoxAnn H. Stalvey. "Bloom's Taxonomy Revisited: Specifying Assessable Learning Objectives in Computer Science." In: *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '08. Portland, OR, USA: ACM, 2008, pp. 261–265.

[51] Errol Thompson, Andrew Luxton-Reilly, Jacqueline L. Whalley, Minjie Hu, and Phil Robbins. "Bloom's Taxonomy for CS Assessment." In: *Proceedings of the Tenth Conference on Australasian Computing Education - Volume 78*. ACE '08. Wollongong, NSW, Australia: Australian Computer Society, Inc., 2008, pp. 155–161. URL: http://dl.acm.org/citation.cfm?id=1379249.1379265.

[52] Alan M. Turing. "On computable numbers, with an application to the Entscheidungsproblem." In: *J. of Math* 58.345-363 (1936), p. 5.

[53] Scott A. Turner, Ricardo Quintana-Castillo, Manuel A. Pérez-Quiñones, and Stephen H. Edwards. "Misunderstandings About Object-oriented Design: Experiences Using Code Reviews." In: *SIGCSE Bull.* 40.1 (Mar. 2008), pp. 97–101.

[54] Maarten W Van Someren, Yvonne F Barnard, Jacobijn AC Sandberg, et al. *The think aloud method: A practical guide to modelling cognitive processes*. Vol. 2. Academic Press London, 1994.

[55] Wikipedia. *Sudoku*. URL: https://en.wikipedia.org/wiki/Sudoku (visited on 09/17/2015).

[56] Chris Wilcox. "The Role of Automation in Undergraduate Computer Science Education." In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE '15. Kansas City, Missouri, USA: ACM, 2015, pp. 90–95.

Appendices

Appendix A

Participant Information Sheet

Forespørsel om deltagelse i forskningsprosjekt

Undersøkelse av underliggende misforståelser av faglig karakter bak kodefeil i introduksjonsemner innen programmering

Bakgrunn og formål

Undersøkelsen er en del av en masteroppgave ved Institutt for informatikk (Ifi)ved Universitetet i Oslo. Hensikten med undersøkelsen er å øke kunnskap om hvilke underliggende misforståelser som ligger bak ulike kodefeil som gjøres av studenter og hvilke tiltak som kan benyttes for å redusere forekomsten av misforståelsene.

Programkode forteller lite om forfatterens hensikt og tanker. Det gjør det vanskelig for rettere og undervisere å identifisere hva det er en student misforstår og hvor det eventuelt er hull i kunnskapen om faget.

Målet med undersøkelsen er:

1. Å identifisere noen av de vanligste kodefeilene studenter gjør i kurset.
2. Å identifisere noen av de mest vanlige misforståelsene som er årsak til de overnevnte kodefeil
3. Forsøke nye tiltak for å redusere forekomsten av de identifiserte misforståelsene

Undersøkelsen er rettet mot studenter ved kurset INF1010 - Objektorientert programmering, holdt av Ifi.

Hva innebærer deltakelse i studien?

Studien involverer først analyse av utvalgte besvarelser på noen av de obligatoriske oppgavene i kurset. Dersom besvarelsen inneholder kodefeil som er relevante for studien vil studenten som skrev besvarelsen inviteres til å delta på et personlig intervju.

Hensikten med intervjuet er å avdekke hva som er årsaken til at vedkommende fikk en slik feil og identifisere misforståelser av faglig karakter. Spørsmålene vil omhandle besvarelsen på den obligatoriske oppgaven og hvordan man gikk frem for å løse den. Intervjuet vil tas opp med diktafon.

Dersom det viser seg hensiktsmessig vil deltagerne i studien inviteres til å delta i en del av studien der en fra prosjektgruppen observerer dem mens de løser en obligatorisk oppgave. Hensikten med observasjonen vil være å få utdypt dataene som samles inn i intervjuet.

Informasjonen fra studien vil kunne brukes til tester med formål om å redusere forekomsten av kodefeil hos studenter.

Hva skjer med informasjonen om deg?

Alle personopplysninger vil bli behandlet konfidensielt og vil ikke få innvirkning på resultater i kurset eller være tilgjengelig for personer med undervisning- eller retteransvar for deltagerne.

Personopplysninger vil kun være tilgjengelige for masterstudenten (Mathias J. Johansen) og veileder (Ragnhild K. Runde). Dataene fra studien, inkludert transkriberte lydopptak, blir anonymisert og erstattet med referansenøkkel som lagres separat fra resten av dataen. Lydopptak vil transkriberes og deretter slettes.

Prosjektet skal etter planen avsluttes 31/12-2015. Alle personopplysninger vil da slettes og alle data vil være anonyme. Personopplysninger vil ikke være med i masteroppgaven og deltagere vil ikke være mulig å gjenkjennes ut fra den.

Frivillig deltakelse

Det er frivillig å delta i studien, og du kan når som helst trekke ditt samtykke uten å oppgi noen grunn. Dersom du trekker deg, vil alle opplysninger om deg bli anonymisert. I likhet med opplysninger som avdekkes i løpet av studien vil ikke hvorvidt man deltar i studien eller velger å trekke seg bli kommunisert til personer med undervisning- eller retteransvar for deltagerne.

Dersom du har spørsmål til studien, ta kontakt med Ragnhild Kobro Runde via epost (ragnhild.runde@ifi.uio.no) eller eventuelt telefon (22840144/94436221).

Studien er meldt til Personvernombudet for forskning, Norsk samfunnsvitenskapelig datatjeneste AS.

Samtykke til deltakelse i studien

Jeg har mottatt informasjon om studien, og er villig til å delta

---

(Signert av prosjektdeltaker, dato)

☐ Jeg samtykker at mine besvarelser på obligatoriske oppgaver brukes i studien

☐ Jeg samtykker til å delta i intervju

☐ Jeg samtykker til å delta i obervasjon

☐ Jeg samtykker til at mine eksamensresultater i INF1010 brukes i studien

Appendix B

Interview Guide

Intervjuguide

March 23, 2015

1 Hensikt

Hensikten med intervjuet er å kartlegge hvordan studenten har gått frem for å løse en oppgave. Før intervjuet gjennomføres må intervjuer ha satt seg inn i intervjuobjektets besvarelse av oppgaven og identifisert kodefeil som er relevante for studien. I tillegg skal intervjuer ha lest grundig gjennom oppgaveteksten og identifisert mulige tekstlige årsaker til misforståelser om hensikten og gjennomføringen av oppgaven.

Feilene vi ønsker å finne årsaken bak er av følgende type:

- Feil i iterator:
  - Feil bruk av Iterable vs. Iterator
  - Metoder som itererer igjennom hele containeren i stedet for å hente ett og ett objekt
  - Pekerkrøll
- Feil i generiske typer:
  - Manglende bruk av metoder som kommer gjennom restricted types (compareTo og samme)
  - Tegn som typer på at studenten tror navnet på den generiske typen er "vesentlig"
- Fil i lenket liste:
  - Pekerkrøll ved innsetting og sortering
  - Feil bruk av metoden compareTo

2 Spørsmål vi skal besvare:

Innen intervjuet er over bør følgende ting kunne svares på for alle identifiserte kodefeil:

☐ Var studenten klar over at det var feil i koden?

☐ Hvordan gikk studenten frem for å løse problemet der feilen er?

☐ Hva er årsaken til at studenten valgte å løse det slik?

☐ Hvor viktig del av oppgaven anså studenten at delen feilen var i til å være?

I tillegg vil følgende informasjon være interessant på generell basis:

☐ Hvilket ferdigheter/hvilken kunnskap tror studentene oppgaven tester ut?

3 Off topic:

Følgende ting er ikke relevante for undersøkelsen og skal ikke spørres om:

- Studentens kjønn, alder osv.
- Studentens studieprogram
- Studentens tidligere og videre studieplan
- Studentens besvarelse av andre oppgaver enn den intervjuet omhandler. Med mindre oppgavene bygger på hverandre.
- Opplysninger om tredje person (faglærer, gruppelærer, medstudenter)
- Informasjon om undervisning fra spesifike gruppelærere eller faglærere
- Informajson om utsatt innlevering av oppgaven som bla. kan inkludere informasjon om helseforhold (sykemelding/egenmelding)

4 Struktur

4.1 Introuksjon

Presenter deg selv, bakgrunnen din og arbeidet du gjør. Forklar hensikten med studien for å minne studenten om hva vi ønsker å oppnå med intervjuet.

Fortell studenten at han/hun kan prate fritt og at han/hun ikke trenger å vente på noe spørsmål, men bare fortsette å forklare oppgaven slik det føles naturlig.

Sørg for å gjøre det klart at ingen opplysninger fra dette intervjuet vil påvirke hans/hennes resultater i kurset eller å noe vis komme i hende de som underviser eller retter oppgavene hans/hennes.

Forklar bruken av lydopptaker. Dette er for at intervjuer skal få med seg alle detaljer i samtalen og skal transkriberes i etterhånd. Forklar at ingen andre enn prosjektgruppen kommer til å høre lydbåndene og at de slettes etter at de har blitt transkribert og anonymisert.

Forklar at hvis det er noe studenten ikke ønsker å svare på eller ønsker å trekke seg fra intervjuet så kan han/hun gjøre det når som helst.

Spør om studenten har noen spørsmål før du starter.

4.2 Start

Start intervjuet med å spørre studenten hva han/hun tror var det viktige å teste i denne oppgaven.

Spør studenten om han/hun vet om noen feil i den innleverte besvarelsen. Hvis ja: be studenten om å forklare hvorfor han/hun tror det er en feil. Hvis nei: be studenten forklar hvordan han/hun gikk frem for å løse oppgaven steg for steg.

Eks: Når du startet å jobbe med obligen. Hvor startet du hen da?

4.3 Per tema/deloppgave/feil

For et tema i oppgaven eller en feil studenten går innpå så bør du sørge for at han/hun forklarer i tilstrekkelig detalj hvordan han/hun jobbet for å løse det.

Følgende spørsmål kan være greie hvis studenten stopper opp eller ikke går dypt nok inn på temaet til å få besvart spørmålene i seksjon 2:

- For å løse denne oppgaven, hvor startet du?
- Hva er problemet her?
- Hvordan løste du problemet?
- Hvorfor gikk du for den løsningen (kontra en annen løsning)?
- Ser du noen måte din løsning kan skape problemer/feil i programmet?
- Hvorfor tror du at du bes om å bruke løsningstype X i stedet for Y her?

Utenom det er det ønskelig å la studenten styre samtalen. Intervjuers oppgave er å sette i gang samtalen og guide studenten ved behov. Vi har lite grunnlag for å gjøre antagelser om rekkefølge noe er gjort i og er interessert i å vite hva studenten anser som mest relevant i sin besvarelse.

Vær oppmerksom på om studenten forklarer seg med grunnlag i hva oppgaveteksten ver om eller andre kilder til hva poenget er (f.eks andre studenter eller muntlig beskjed fra faglærere/gruppelærere). Kilden til misforståelsen kan være interessant.

Hvis studenten går off topic skal du lede intervjuet tilbake til temaet, studentens besvarelse.

5 Avslutning

Spør studenten om han/hun var inne til rettetime før besvarelsen ble levert og om det har blitt endret etter det.

Spør også hvorvidt han/hun gjorde dette alene eller som del av et parprogrammeringsteam.

Forklar studenten hva det videre arbeidet med analyse vil foregå hvordan intervjuene vil presenteres i masteroppgaven. Hvis studenten ønsker så kan han/hun bli satt på en liste for å motta evt publisert materiale.

Spør om studenten har noen flere spørsmål nå som intervjuet er over.

6 Etter intervjuet

Skriv ned eller lag memo med diktafonen med tanker om hvordan intervjuet gikk. Tanker om faglige misforståelser studenten kan ha hatt og hvordan det lenker til resten av arbeidet. Vurder om intervjuet kan ha blitt påvirket av atmostfære eller at studenten har vært nervøs, sliten, trøtt osv.

Appendix C

Approval letter from the Data Protection Official for Research

Norsk samfunnsvitenskapelig datatjeneste AS
NORWEGIAN SOCIAL SCIENCE DATA SERVICES

Harald Hårfagres gate 29
N-5007 Bergen
Norway
Tel: +47-55 58 21 17
Fax: +47-55 58 96 50
nsd@nsd.uib.no
www.nsd.uib.no
Org.nr. 985 321 884

Ragnhild Kobro Runde
Institutt for informatikk Universitetet i Oslo
Postboks 1080 Blindern
0316 OSLO

Vår dato: 11.03.2015 Vår ref: 42265 / 3 / KH Deres dato: Deres ref:

TILBAKEMELDING PÅ MELDING OM BEHANDLING AV PERSONOPPLYSNINGER

Vi viser til melding om behandling av personopplysninger, mottatt 15.02.2015. Meldingen gjelder prosjektet:

42265	*Undersøkelse av underliggende misforståelser av faglig karakter bak kodefeil i introduksjonsemner innen programmering*
Behandlingsansvarlig	*Universitetet i Oslo, ved institusjonens øverste leder*
Daglig ansvarlig	*Ragnhild Kobro Runde*
Student	*Mathias Johan Johansen*

Personvernombudet har vurdert prosjektet og finner at behandlingen av personopplysninger er meldepliktig i henhold til personopplysningsloven § 31. Behandlingen tilfredsstiller kravene i personopplysningsloven.

Personvernombudets vurdering forutsetter at prosjektet gjennomføres i tråd med opplysningene gitt i meldeskjemaet, korrespondanse med ombudet, ombudets kommentarer samt personopplysningsloven og helseregisterloven med forskrifter. Behandlingen av personopplysninger kan settes i gang.

Det gjøres oppmerksom på at det skal gis ny melding dersom behandlingen endres i forhold til de opplysninger som ligger til grunn for personvernombudets vurdering. Endringsmeldinger gis via et eget skjema, http://www.nsd.uib.no/personvern/meldeplikt/skjema.html. Det skal også gis melding etter tre år dersom prosjektet fortsatt pågår. Meldinger skal skje skriftlig til ombudet.

Personvernombudet har lagt ut opplysninger om prosjektet i en offentlig database, http://pvo.nsd.no/prosjekt.

Personvernombudet vil ved prosjektets avslutning, 31.12.2015, rette en henvendelse angående status for behandlingen av personopplysninger.

Vennlig hilsen

Katrine Utaaker Segadal

Kjersti Haugstvedt

Kontaktperson: Kjersti Haugstvedt tlf: 55 58 29 53

Dokumentet er elektronisk produsert og godkjent ved NSDs rutiner for elektronisk godkjenning.

Vedlegg: Prosjektvurdering
Kopi: Mathias Johan Johansen mathiajj@ifi.uio.no

Personvernombudet for forskning

Prosjektvurdering - Kommentar

Prosjektnr: 42265

Utvalget informeres skriftlig om prosjektet og samtykker til deltakelse. Informasjonsskrivet er godt utformet.

Personvernombudet legger til grunn at studentene etterfølger Universitetet i Oslo sine retningslinjer for datasikkerhet.

Forventet prosjektslutt er 31.12.2015. Ifølge prosjektmeldingen skal innsamlede opplysninger da anonymiseres. Anonymisering innebærer å bearbeide datamaterialet slik at ingen enkeltpersoner kan gjenkjennes. Det gjøres ved å:
- slette direkte personopplysninger (som navn/koblingsnøkkel)
- slette/omskrive indirekte personopplysninger
- slette lydopptak.

Appendix D

INF1010 Mandatory Assignment 5

Om obligatorisk oppgave 5, 6 og 7 i INF1010, våren 2015: "Leger og resepter"
Versjon 1,0

Innledning.

Du skal jobbe med en problemstilling omkring leger og resepter i 4 uker fra 19. februar til 25. mars og du kan til sammen få maksimum 8 poeng. Den første delen av oppgaven (oblig 5), som du skal jobbe med de første to ukene (maks 2 poeng), skal leveres senest 4. mars kl 10:00. Den 26. februar frigis den andre del av oppgaven (oblig 6). Denne skal leveres senest 18. mars kl 10:00 og gir maks 3 poeng. Den siste delen (oblig 7) frigjøres 5. mars. Den skal leveres senest 25. mars kl 10:00, og har også maks 3 poeng.

Siden disse tre oppgavene bygger på hverandre er det bare mulig å hoppe av underveis. Det er ikke mulig å hoppe på, dvs. at du må ha godkjent 5 for å få godkjent 6, og du må ha godkjent 6 for å få godkjent 7. Du har godkjent en oppgave om du får ett eller flere poeng.

Obligene 6 og 7 skal løses vha. systemutviklingsmetoden "Parprogrammering".
Du MÅ melde deg på oblig 6 og 7 innen 25. februar. Informasjon om dette finner du under "oblig 6 og 7" på kursets hjemmeside.

I hovedsak består oblig 5 av å lage et klassehierarki, oblig 6 av å lage en del beholdere, og oblig 7 av å sette det hele sammen til slutt til et ordrestyrt program. I oblig 6 og oblig 7 må du kanskje gå tilbake til klassehierarkiet i oblig 5 og gjøre små forandringer.

Obligatorisk oppgave 5, INF1010, våren 2015, maks 2 poeng.
Innlevering senest onsdag 4. mars kl. 10:00

KLASSEHIERARKIENE til Leger og Resepter m.m.

Hensikten med denne oppgaven er at du skal lære deg å skrive klassehierarkier med subklasser og grensesnitt (interfaces). Det er ingen generiske klasser eller generiske interfacer i oblig 5. Ikke gjør oppgaven for stor eller for vanskelig. Som beskrevet over må du kanskje utvide noen av klassene du skriver i oblig 5 når du ser hva obligene 6 og 7 trenger. Du skal derfor foreløpig bare ta med de variable og metodene som det eksplisitt blir bedt om her. Hvis du synes noe er uklart, gjør dine egne fornuftige presiseringer.

Legemidler

Et legemiddel har et navn, et unikt nummer og en pris. Når nye legemidler registreres gis de et nytt løpende (unikt) nummer som starter på 0.
Et legemiddel er enten av type A, narkotisk, eller av type B, vanedannende, eller av type C, vanlige legemiddel. For alle legemidler må vi kunne vite hvor mye virkestoff (mg) det inneholder totalt.

Det er stor forskjell på legemidler av disse tre typene, men i denne oppgaven skal vi bare ta hensyn til:
Legemidler av type A har et heltall som sier hvor sterkt narkotisk det er.
Legemidler av type B har et heltall som sier hvor vanedannende det er.
Legemidler av type C har ingen nye egenskaper (annet enn klassens navn).

I tillegg til egenskapene beskrevet over kommer legemidler enten som piller eller som mikstur (flytende). Disse egenskapene skal du beskrive vha. grensesnitt (interface).

For enkelhets skyld skal vi for piller bare vite hvor mange piller det er i en eske og hvor mye virkestoff det er i hver pille. For mikstur skal vi vite hvor mange cm3 det er i en flaske og hvor mye virkestoff det er i hver cm3.

Resepter

En resept har et unikt nummer som starter på 0 med første resept som opprettes. En resept inneholder en peker til et legemiddel, en peker til den legen som har skrevet ut resepten, og nummeret til den personen som eier resepten (se nedenfor om leger og personer).
En resept har et antall ganger som er igjen på resepten (kalles "reit"). Hvis antall ganger igjen er null, er resepten ugyldig.

Noen resepter er blå, andre er hvite. Vi skal ta høyde for at det er stor forskjell på hvite og blå resepter, men igjen skal vi gjøre en forenkling og si at bare prisen er forskjellig: Blå resepter er sterkt subsidiert, og for enkelhets skyld sier vi her at de er gratis.

Leger

En lege har et unikt navn.

I oblig 6 skal vi kunne finne en lege basert på navn. Klassen Lege skal derfor implementere grensesnittet (interfacet) Lik der sammenlikningen (likheten) skal skje med legens navn.

Grensesnittet Lik inneholder en metode kalt "samme" som har som parameter en String og returnerer sann eller usann. Dette grensesnittet kan f.eks. brukes til å finne om et objekt som inneholder et navn (String) har samme navn som parameteren til metoden.

Noen leger har avtaler med kommunen der de jobber (fastleger). Dette å ha en avtale med kommunen er noe ikke bare leger kan ha. Denne egenskapen skal derfor beskrives med et grensesnitt (interface). For alle som har en avtale med kommunen skal vi kunne få tak i et avtalenummer.

Personer

Personer har et navn og et fødselsnummer (11 siffer) og en adresse. Adressen skal være et veinavn og nummer (som til sammen lagres i en String) og et firesifret postnummer. Når en ny person registreres gis personen i tillegg et nytt løpende (unikt) nummer som starter på 0 med første personen som programmet oppretter.

Oppgave 5.1
Tegn opp klassehierarkiene beskrevet over. Ta også med alle grensesnitt (interface). I denne tegningen skal du ikke ta med data og metoder.

Oppgave 5.2
Skriv programmene for alle klassene og grensesnittene (interface) beskrevet over.
Skriv konstruktører som initialiserer alle variable og konstanter i alle klasser.

Du skal lever inn tegningen fra oppgave 5.1 og programmene (kildekoden) til alle klassene og grensesnittene (interface) fra oppgave 5.2. Du skal følgelig ikke levere noe fullstendig, kjørbart program i denne oppgaven.

Slutt obligatorisk oppgave 5.

Appendix E

INF1010 Mandatory Assignment 6

Obligatorisk oppgave 6 i INF1010, våren 2014: "Beholdere til Leger og resepter"
Versjon 1,0. Frigis 26, februar, 3 poeng. Leveres innen kl 10:00 onsdag 18. mars.

I denne oppgaven skal dere vise at dere kan bruke grensesnitt, abstrakte klasser, generiske klasser, subklasser, lenkelister og tabeller.
Dere skal programmere en del beholdere som skal brukes i oblig 7. Mange av disse beholderene skal kunne brukes til andre formål, men for å ikke gjøre oblig 6 for abstrakt er det kanskje lurt å ha bruken i oblig 7 klart for øyet hele tiden.
På gruppene vil dere få hjelp til å lage iteratorer. Når dere implementerer iteratorer behøver dere ikke implementere remove().

De beholderene dere skal programmere i oblig 6 skal i oblig 7 ta vare på alle legemidler, resepter, leger og personer. I tillegg skal en lege ha en beholder over alle reseptene vedkommende har skrevet ut, og en person skal ha en beholder som inneholder alle personenes resepter. Legemidlene skal lagres i et objekt av klassen Tabell, reseptene skal lagres i et objekt av klassen EnkelReseptListe, legene skal lagres i et objekt av SortertEnkelListe og personene skal lagres i et objekt av klassen Tabell. Beholderen som inneholder en leges resepter skal være av klassen EldsteForstReseptListe, mens beholderen som inneholder en persons resepter skal være av klassen YngsteForstReseptListe.

Les gjennom hele oppgaven før du starter å svare på noen spørsmål. Du må ha klar oversikt over resten av oppgave 6 før du besvarer oppgavene 6.1.

Oppgave 6.1.a. Tegn opp klassehierarket for beholderne.
Ikke ta med navn på metoder og variable

Oppgave 6.1.b. Tegn opp datastrukturen.
Tegn alle beholderne, noen legemiddel-objekter, noen lege-objekter, noen person-objekter og noen resept-objekter. La det komme klart frem at en resept er med i mange beholdere. Tegn inn ”public” metodene (grensesnittet) i alle beholderene. Ikke ta med andre metoder. Når det er flere like objekter behøver du bare tegne inn all variable i ett objekt.

I oppgave 6.2 og 6.3.a frigjør vi oss fra leger, resepter mm. og definere generelle grensesnitt og klasser for beholdere:

Oppgave 6.2. Grensesnitt (interface) for beholdere

Skriv programmet for det generiske grensesnittet AbstraktTabell. Det skal ikke være noen restriksjoner på hva slags elementer den abstrakte tabellen skal kunne inneholde.
AbstraktTabell beskriver en beholder og du skal kunne:
- sette et objekt inn i tabellen på en oppgitt plass (indeks). Metoden returnerer sann eller usann avhengig om operasjonen gikk bra eller ikke.
- finn et objekt basert på en indeks.
- iterere over listen.

Skriv programmet for det generiske grensesnittet AbstraktSortertEnkelListe. En slik liste skal bare kunne inneholde elementer som implementerer grensesnittene Comparable (med seg selv) og Lik. En slik liste skal kunne:
- sette inn et nytt element (i sortert rekkefølge, minste først).
- finne et element basert på en nøkkel av typen String
- itereres over, slik at innholdet kan bli listet opp i sortert rekkefølge, minste først.

Oppgave 6.3. Klasser for beholdere

a) Generiske klasser

Skriv den generiske klassen Tabell som implementerer AbstraktTabell.
Klassen skal lagre alle elementene i en array, og arrayens lengde skal oppgis som parameter til konstruktøren.

Hvis du har lyst og tid: Når du setter noe inn i Tabellen og det ikke er plass, skal du lage en ny array som er lang nok (innenfor rimelighetens grenser), og så kopiere alle elementene over til den nye arrayen.

Skriv den generiske klassen SortertEnkelListe som implementerer AbstraktSortertEnkelListe som en enveisliste. Du skal programmere denne listen selv, ikke bruke klasser fra Java-biblioteket.

b) Ikke generiske klasser

Skriv klassen EnkelReseptListe. Klassen EnkelReseptListe skal inneholde en envisliste med en peker til første og en peker til siste element i listen. Klassen skal kunne ta vare på resepter, og en resept må kunne være med i flere objekter av denne klassen. Metodene i klassen skal kunne sette inn en resept og finne en resept basert på reseptnummeret. Hvis resepten som det letes etter ikke finnes i listen, skal det kastes et unntak. Skriv også en iterator over listen. Igjen skal du programmere denne listen selv, ikke bruke klasser fra Java-biblioteket.

Skriv subklassene EldsteForstReseptListe og YngsteForstReseptListe. Når du itererer over den første klassen skal du starte med den eldste resepten (den som ble satt inn først) og gå mot yngre (de som ble satt inn sist). Når du itererer i den andre klassen, skal du starte i den yngste enden.

Hint: Forskjellen på de to subklassene til klassen EnkelReseptListe kan være bare metoden som setter inn en resept.

Utfordring: I både SortertEnkelListe og EnkelReseptListe skal du skrive en iterator. Kan du klare å bruke den samme iteratoren i begge klassene? Da må de to listene kanskje ha de samme nodene?

Oppgave 6.4. Lag enkle enhetstester for alle beholderene.

Skriv et lite testprogram for hver av klassene Tabell, SortertEnkelListe, EldsteForstReseptListe og YngsteForstReseptListe. Prøv å lage programmene slik at både vanlige tilfeller og noen spesialtilfeller blir testet.

Innleveringen på denne oppgaven er tegningene fra oppgave 6.1, alle grensesnittene og klassene, og de fire kjørbare programmene fra oppgave 6.4

Appendix F

INF1010 Mandatory Assignment 9-11

INF1010 2015 — Obligatorisk oppgave 9, 10 og 11 — SUDOKU

Versjon 26. mars. Ved behov for presiseringer i del 10 og 11, kan det komme ny versjon 16. april, men ingen endringer i selve oppgaven.

Denne obligatoriske oppgaven skal løses individuelt. Det betyr at alt som leveres for retting skal du selv ha skrevet inn. Det er ikke lov å ta inn kode som er laget av andre. Hverken andre studenter, fra nettet eller andre kilder. Oppdages slik kode i noe som er levert vil det bli oppfattet som forsøk på fusk og fulgt opp i tråd med regelverket om forsøk på fusk.

Denne obligen dekker følgende deler av pensum (stikkord): objektorientert programmering, rekursjon, GUI, lenkelister, ...

Obligen er delt inn i tre innleveringer (9, 10 og 11), slik at det er mulig å få tilbakemeldinger underveis. Oppgave 9 og 10 er igjen delt i to, slik at arbeidet fordeles på en 5-ukers periode med ca. 2 poengs arbeid pr uke. Tilsammen skal du ende opp med et program som kort fortalt løser sudokuoppgaver. Men før vi beskriver de enkelte delene, tar vi en gjennomgang av hele oppgaven for oversiktens skyld:

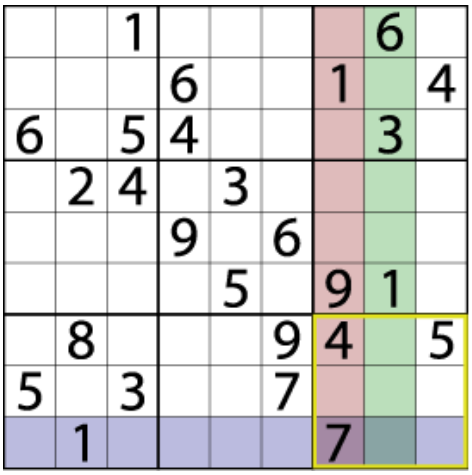
Hva er sudoku?

Se denne wikipediartikkelen:

`http://no.wikipedia.org/wiki/Sudoku`. (For mange fler detaljer, gå til den engelske wikipediaartikkelen `http://en.wikipedia.org/wiki/Sudoku`.)

Sudokubrettet

Et sudokubrett består av n × n ruter. Vi bruker følgende begreper i oppgaven:



- **rute** er feltet som det kan stå ett tall (eller én bokstav) i.
- **brett** er alle $n \times n$ ruter.
- **rad** er en vannrett (fra venstre mot høyre på brettet) rekke med n ruter.
- **kolonne** er en loddrett (ovenfra og nedover) rekke med n ruter.
- **boks** er flere vannrette og loddrette ruter, markert med tykkere strek i oppgavene, ofte med vekslende bakgrunnsfarge; i 9×9-sudoku er en boks på 3×3 ruter, mens i lynsudoku består en boks av 2×3 ruter.

Brettet ovenfor er et 9×9-brett ($n = 9$), dvs. rader, kolonner og bokser har alle 9 ruter. Det er også 9 rader, 9 kolonner og 9 bokser i brettet. 2 kolonner er markert med rød og grønn bakgrunnsfarge. Nederste (niende) rad er markert med fiolett, mens boksen nederst til høyre er rammet inn med gult. Det er vanlig å si at øverste rad er første rad, mens kolonnen lengst til venstre er første kolonne.

Disse begrepene vil i javaprogrammet finnes igjen som objekter. F.eks. vil hele brettet representeres av et objekt som bl.a. ineholder en todimensjonal array av ruter. Hver rute vil igjen inneholde informasjon om hvilken rad, kolonne og boks ruta ligger i.

Du skal lage programmet så generelt at det også kan løse sudokubrett som ikke har kvadratiske bokser. Brettet ovenfor er eksempel på et brett med kvadratiske bokser. Merk at 9×9 ikke er noen øvre grense for størrelsen på brettet. Eksempel på et ikke-kvadratisk brett ser du nedenfor (6×6).

En **sudokuoppgave** er et delvis utfylt brett som kan ha tre løsningsmuligheter:

1. én løsning (slike finner vi i aviser, bøker og blader)
2. ingen løsning (tallene er plassert slik at det ikke finnes en løsning)
3. flere løsninger (for få forhåndsutfylte tall)

Et tomt 9×9 brett har 6.670.903.752.021.072.936.960 løsninger.

Hovedtappene under utførelsen av programmet

- lese inn brettet
- opprette en datastruktur som tilsvarer det innleste brettet
- finne alle løsninger og lagre disse i en beholder
- skrive ut løsningene til skjerm og fil
- skrive ut løsningene én etter én, når brukeren trykker på en knapp

Men utviklingen trenger ikke skje i denne rekkefølgen. Mer avansert GUI, f.eks. kan utvikles uavhengig av resten. Det samme gjelder innlesing fra fil, og beholderen som lagrer løsninger. Datastruktur og metoden som finner løsninger, derimot, henger tett sammen. Vi anbefaler at man begynner med datastrukturen. Dette er av erfaring den delen man trenger lengst tid på. Derfor er løsningen av denne skilt ut som det første du skal gjøre.

Filformatet vi bruker

For å lagre sudokuoppgaver har vi derfinert et eget filformat. Dette 6×6-brettet beskrives av filen til høyre. Merk at filen ikke har blanke tegn i seg:

		3	6		
	2				4
5				6	
	3				5
3				1	
		1	4		

```
2
3
..36..
.2...4
5...6.
.3...5
3...1.
..14..
```

Første tall (2) er antall rader i hver boks, og neste tall (3) er antall kolonner i hver boks. $2 \times 3 = 6$. Brettet er da $6 \times 6 = 36$ ruter. 6 er også antall ruter i kolonne, boks og rad. Så følger selve brettet. Punktum (.) betyr tom rute. Hvis vi trenger mer enn 10 siffer, bruker vi $A = 10$, $B = 11$, osv. (Se sudokusiden for eksempler).

Programmodulen som leser fra fil og oppretter brettet kan løses tidlig eller senere. For å få poeng for denne delen, må den seinest tas med i leveringen

av oblig 10.

Del 9a—Les inn fra fil og datastruktur for brett og ruter

Denne delen kan du få rettet og testet i perioden 8. april - 28. april.

Størrelsen brettet og selve oppgaven (de sifrene som alt er fylt inn) leses fra fil.

Det er mulig å gjøre selve innlesningsdelen etter del 9b, men for å ha data å jobbe med er det greit å lage en metode som leser inn sudokuoppgaven fra fil med en gang. Den første oppgaven består derfor av å lage en metode som løper gjennom filen, bestemmer størrelsen på brett og boks og oppretter datastrukturen for brettet.

Klassen Brett må inneholde info om alle rutene i sudokubrettet. Info om hver rute lagres i objekter av klassen Rute.

Klassen Rute må inneholde en verdi (husk at noen av rutene ikke har en verdi ennå). For hver rute du leser inn; opprett et objekt av klassen Rute og lagre verdien. Lagre så hver rute i brettet (f.eks. i et array).

Lag så en utskriftsmetode som løper gjennom rutene i brettet ditt og skriver ut verdiene til skjerm. Utskriften fra eksempelfilen i oppgaven bør se sånn ut:

```
. . 3 6 . .
. 2 . . . 4
5 . . . 6 .
. 3 . . . 5
3 . . . 1 .
. . 1 4 . .
```

Oppsummering: Lag en metode lesFil() som leser inn en fil og lagrer infoen i et objekt av klassen Brett. Lag så en utskriftsmetode som skriver ut det innleste brettet til skjerm.

Del 9b—utvid datastrukturen. *Leveringsfrist 22. april*

Denne delen kan du få rettet og testet i perioden 8. april - 28. april.

For å gjøre det mulig å løse sudokubrettet må klassen Rute i tillegg til verdi inneholde info om hvilken rad, kolonne og boks den tilhører.

Dette gjøres ved at *hver rute* inneholder pekere til ett objekt av klassen Boks, ett objekt av klassen Rad og ett objekt av klassen Kolonne. Ruter i samme rad skal peke på samme radobjekt osv.

Disse klassene skal brukes for å bestemme hvilke verdier som er «opptatt» i raden, kolonnen og boksen. Klassen rad, kolonne og boks må derfor inneholde info om hvilke verdier som allerede er tatt i den raden/kolonnen/boksen.

Lag en metode

```
void delInnRuter()
```

som oppretter riktig antall rader, kolonner og bokser, for så å få hvert ruteobjekt i brettet til å peke på sine respektive rader, kolonner og bokser. For at inndelingen i bokser skal bli riktig må du ha tatt vare på de 2 første verdiene i fila som sier antall rader i hver boks og antall kolonner i hver boks.

Tegn datastrukturen til ruta i 3. rad og 2. kolonne i figuren under beskrivelsen

av filformatet. *Denne tegningen skal leveres i Devilry hvis din retter ikke har gitt fritak i forbindelse med retting.*

Skriv tilslutt en metode i klassen Rute

```
int[] finnAlleMuligeTall()
```

som returnerer en array med de tallene som er mulige løsningstall i en blank rute. F.eks. skal metoden kalt i øverste rute til venstre i eksempelbrettet på første side returnere en array med tallene 2, 3, 4, 7, 8 og 9.

Oppsummering: Utvid klassen Rute til å inneholde info om rad, kolonne og boks. Lag så metoden int[] finnAlleMuligeTall(). Husk tegning av datastrukturen til en rute.

Tips: Gi hver rad/kolonnoe/boks en unik ID, på denne måten kan du enkelt lage en testutskrift som løper gjennom brettet og skriver ut hvilken rad/kolonne/boks hver rute tilhører. Da ser man fort om man har gjort en feil. *Eller:* Lag en testmetode som løper igjennom alle rutene og som for hver rute skriver ut info om verdi, rad, kolonne og hvilke tall som er mulige løsningstall (ved kall på finnAlleMuligeTall)

Del 10a—løsningsmetoden og utskrift av løsninger

Del 10 og 11 kan bli endret noe ved behov. Det vil ikke bli noen endringer i hva som skal gjøres, men evt. bedre formuleringer, presiseringer og flere detaljer. Det skjer i såfall i ny versjon av obligteksten som publiseres seinest 16. april.

Denne delen kan du få rettet og testet i perioden 22. april - 5. mai

Løsningene skal finnes ved å gå gjennom alle rutene på brettet og prøve alle mulige (lovlige) verdier i hver eneste rute. Dette kalles «rå kraft»-metode («brute force» på engelsk).

Utvid klassen Rute med en metode, fyllUtDenneRuteOgResten, som prøver å sette alle mulige løsningstall i seg selv. Det aller første denne metoden gjør er å kalle på metoden finnAlleMuligeTall. Deretter prøver den å sette alle disse tallene i denne ruta, ett om gangen. For hvert tall som settes i ruta kalles samme metode (fyllUtDenneRuteOgResten) i neste rute (dvs den rett til høyre). Når en vannrett rad er ferdig (det finnes ingen rute rett til høyre), kalles metoden i ruta helt til venstre i neste rad, osv. Når et kall på fyllUtDenneRuteOgResten-metoden i neste rute returnerer, prøver ruta neste tall som enda ikke er prøvd, osv. helt til alle tall er prøvd i denne ruta. Main-metoden starter det hele ved å kalle fyllUtDenneRuteOgResten i den øverste venstre ruta. (Hint: Du kan gjerne lenke sammen alle rutene med en nestepeker, slik at en rute bare kan kalle neste.fyllUtDenneRuteOgResten). Når metoden har funnet en lovlig verdi i den siste ruta (den nederst til høyre) på brettet, er en løsning funnet.

Løsningen(e) skal skrives ut på skjermen.

Hint: Under utviklingen kan det være lurt å først lage et program som genererer alle løsninger for et tomt brett for så senere å utvide med at noen av rutene kan ha forhåndsutfylte verdier. Ikke bruk et tomt brett med mer enn 9×9 ruter, da dette kan ta fryktelig lang tid.

Del 10b—utskrift og lagring av løsninger. *Leveringsfrist 6. mai*

Denne delen kan du få rettet og testet i perioden 9. april - 13. mai.

Filnavnet oppgis som parameter til programmet (på kommandolinja). Filformatet skal være slik som beskrevet over (retter vil teste programmet ditt med andre filer). Hvis det oppgis ett filnavn skal løsningen(e) skrives til skjerm. Hvis det oppgis to filnavn skal oppgaven løses fra den første filen, og løsningen(e) skrives på den andre filen (og ikke skrives til skjerm).

Filformatene for utskrift er beskrevet nedenfor.

Programmet skal inneholde class SudokuBeholder som igjen inneholder de tre offentlige metodene settInn, taUt og hentAntallLosninger. Du kan lage beholderklassen selv (da lærer du best), men kan også gjenbruke klassen fra en tidligere oblig, eller fra Javas API.

Programmet skal finne løsninger og legge dem inn i et objekt av klassen SudokuBeholder. Hvis det finnes flere løsninger enn 2500, skal beholderen holde orden på hvor mange løsninger som er funnet, men ikke ta vare på flere enn 2500 løsninger.

Oblig 11—framvisning av løsninger med vindusbasert GUI. *Leveringsfrist 13. mai*

Denne delen kan du få rettet og testet i perioden 6. - 20. mai.

I denne skal du lage et grafisk brukergrensesnitt (GUI) med Swing og Awt for å kommunisere bedre med brukeren og for å skrive ut løsninger. Altså 2 nye krav:

1. Programmet skal bruke JFileChooser til å finne/velge filen med oppgaven.

2. Et annet vindu skal vise fram løsningen(e) ved å hente den/dem fra beholderen etter at oppgaven er løst. Er det flere enn en løsning vises bare den første fram, men informasjon om hvor mange løsninger som totalt ble funnet.

I den avsluttende delen skal du utvide GUI-et med *lyttende knapper* slik at løsningene (hvis flere) kan vises fram en etter en når brukeren tryker på en knapp. Her kan du også legge inn ande nyttige GUI-funksjoner som f.eks. muligheten til å legge inn en oppgave direkte i vinduet.

Om progresjonen underveis

Retteperiodene under hver del er veiledende. Den enkelte retter kan avvike fra dette, men vær oppmerksom på at **retteperiodene strekker seg etter fristen!** De som ikke møter til retting i retteperioden kan ikke regne med å få tilbakemelding før siste del skal leveres.

Om tråder

I denne oppgaven trenger du ikke programmere med andre tråder enn maintråden og GUI-tråden. Maintråden starter med å opprette et eller flere vinduer for å lese inn data. Disse vinduene skal så lukkes når data er lest inn. Da startes selve løsningsalgoritmen. Når maintråden har funnet en løsning legges denne inn i en beholder. Når alle løsningene er funnet skal maintråden åpne et vindu der brukeren kan be om at en og en løsning blir vist fram ved å trykke på en knapp i vinduet.

Prøv å lage et robust program, dvs. et som ikke kræsjer når filformatet er feil eller noe annet uventet skjer.

Om du synes at noen av disse kravene er urimelige, eller du synes du kan løse oppgaven mer elegant eller bedre på en annen måte, ta det opp med din retter i forbindelse med rettingen av de tidlige delene. En forutsetnig for å gjøre dette, er at man har møtt til retting underveis.

Om formatet for utskrift til skjerm/fil

Når det er få løsninger, kan du bruke samme format for å skrive løsningene ut til skjerm og fil. Punktum byttes ut med løsningstallet. Når det er mange løsninger kan du bruke det alternative utformatet som vises nedenfor. Denne sudokuoppgaven har 28 løsninger, de første 9 av dem er listet opp til høyre i alternativt utformat:

```
2
3
..1..3
.......
....2.
26....
...3..
3..1.2
```

```
1: 421563//653214//134625//265431//512346//346152//
2: 421653//536214//143526//265431//612345//354162//
3: 421653//536214//153426//264531//612345//345162//
4: 421653//635214//513426//264531//142365//356142//
5: 451263//623415//135624//264531//512346//346152//
6: 451263//623514//134625//265431//512346//346152//
7: 451263//632415//513624//264531//125346//346152//
8: 521463//643215//135624//264531//412356//356142//
9: 521643//436215//143526//265431//612354//354162//
```

Retting underveis

Du bør minst ukentlig diskutere med retteren din hvordan du skal løse denne oppgaven. Pass på å hele tiden ha et program som kompilerer og kjører (men som i starten ikke gjør særlig mye). Du bør i det minste kontakte retteren din for å få tilbakemelding etter hver av de fem delene.

I tillegg kan det være greit å få tilbakemeldinger:

• når du har laget main og mange tomme klasser og har en første grove skisse av hele programmet ditt

• når du har bestemt formatet på løsningene slik de skal lagres i sudokubeholderen

• når du har laget en skisse av klassen Rute og dens subklasser

• når du har laget en skisse av klassene Boks, Kolonne og Rad og superklassen til disse klassene.

• når du har laget en skisse av GUI-programmet som henter ut løsninger fra sudokubeholderen og tegner dem ut.

Flere eksempler, eksempler på brett laget med Swing, flere sudokuoppgaver, samt utfyllende informasjon om sudoku finner du på INF1010s Sudoku-side:

`http://heim.ifi.uio.no/inf1010/v15/oblig/91011/sudoku.html`

Her vil vi også legge ut ekstraoppgaver til dem som ønsker flere utfordringer og informasjon om vår programmeringskonkurranse!

Om alternativ datastruktur

Hvis du ønsker å ha variable i klassen Rute som kan brukes til å midlertidig begrense gyldige verdier i denne ruta, eller gjøre andre større endringer i datastruktur, skal du først diskutere dette med din retter og få tillatelse.

Lykke til med programmeringen!

Stein Michael Storleer

Appendix G

Tasks created for observations

Oppgaver

Oppvarmingsoppgave:

I denne oppgaven blir du gitt klassen under:

```
class Bil {
	String modell;
	int vekt;
	int hk;
	Bil(String m, int v, int hk){
		this.modell = m;
		this.vekt = v;
		this.hk = hk;
	}
	public void printInfo(){
		… //skriv denne metoden
	}
}
```

Fullfør koden i metoden printInfo() og skriv et fullstending java-program som lager fire biler og printer ut informasjonen i dem.

Oppgave 1:

I denne oppgaven skal du lage et lite progran som beskriver forholdet mellom ulike typer kjøretøy.

- Alle kjøretøy skal ha en vekt.
- Kjøretøy deles i landkjøretøy og vannkjøretøy
- Alle landkjøretøy har et visst antall hjul og alle vannkjøretøy skal ha en lengde i antall fot
- Kjøretøy kan også være motoriserte. Motoriserte kjøretøy skal vite noe om antall hestekrefter kjøretøyet har.

Lag klasser og interfaces for kjøretøyene sykkel, bil, varebil, taxi, robåt og motorbåt med passende ekstra egenskaper. Merk: det er kun disse klassene det skal kunne lages objekter av.

Oppgave 2:

I denne oppgaven skal du lage en LIFO-kø i form av en lenket liste. Køen skal være generisk og kunne lagre alle ulike klasser.

Skriv klassen med metoder for innsetting, å sjekke om et objekt finnes i køen og får å ta ut et objekt (tips: du trenger kun å kunne ta ut det siste objektet som ble satt inn i køen).

Dersom du skulle ha tid. Gjør køen itererbar ved hjelp av grensesnittene `Iterable` og `Iterator`