Bachelor thesis
Computing Science

Radboud University

The usefulness of automated, style-related feedback to object-oriented programming students

Author:
Rick van der Wal
s1005618

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

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

June 25, 2020

Abstract

To help students learn about code quality, the object-oriented programming course at the Radboud University deployed a new tool that automatically gives feedback on style-related aspects in submitted Java programs. The software performs generic style checks, such as the use of consistent naming conventions, but also more specific feedback based on the assignments, for example to see if certain classes are properly decomposed. This thesis investigates how useful such a tool can be to the students, by determining its accuracy, how the students make use of it, and how they value the feedback. We conducted semi-structured interviews with eight first-year students, which were held in three iterations, distributed evenly over the duration of the course. After each iteration, we improved the tool based on the students' feedback. Finally, after performing qualitative analysis on the results, we concluded that the tool is accurate and quite useful to the students.

Contents

Chapter 1

Introduction

1.1 Problem Description

One important aspect of software development is knowing how to write high-quality source code. It has been shown that low-quality software increases the amount of time spent on adding new features, debugging, and maintaining code in the long run (Akour & Falah, 2016). Therefore, investing extra time and effort into developing high-quality code is likely worth it. However, when teaching computing science students how to program, code quality often does not get enough attention (Wiese, Yen, Chen, Santos, & Fox, 2017). This can be due to to the fact that some aspects of quality are subjective or prone to change, or because teaching beginners about its importance is difficult, as they have not yet worked on larger projects where code quality is vital.

For this purpose, the object-oriented programming course at the Radboud University has deployed Personal Prof, a tool that automatically gives feedback on the quality of source code. Whenever a student hands in their assignment, the tool analyzes the Java code and identifies common quality issues such as missing access modifiers or inconsistent naming conventions. It also runs specific checks for each of the assignments, allowing it to comment on issues such as missing or redundant classes, or on dependencies that should be moved elsewhere. The generated feedback is then sent back to the student within about five minutes. They are allowed to adapt their assignment as often as they want before it gets graded, so they can choose to resolve all indicated quality flaws. Since this software was introduced for the first time this year, we would like to investigate its usefulness to the students.

1.2 Purpose

If Personal Prof works as intended, it can trigger the students to put extra thought into the quality of their code, and help them solve their quality-related issues. This could prevent the aforementioned problems that might arise in low-quality code. The purpose of this research is to determine how effective this approach is.

1.3 Approach

We analyzed the Personal Prof usage of eight students from the object-oriented programming course. These students were split up into three groups, each of which took part in one of three iterations, distributed evenly over the duration of the course. Each iteration, the corresponding students were interviewed about their views on code quality and the feedback they received. Moreover, we assessed the code quality of their assignment from that week. The responses to the interview were used to determine the students' knowledge about code quality aspects, their usage of the tool, and their opinions on the feedback loop. The assessments were used to determine how accurately Personal Prof can detect quality issues. At the end of each iteration, we analyzed the data and used the intermediate results to adapt the tool, in order to resolve any issues the students had while using it, and to improve the tool for later iterations.

1.4 Overview

Chapter 2 provides all background information needed for this thesis, including other papers' findings on the subject. Chapter 3 describes and decomposes the research question this thesis attempts to answer. A detailed description of how this research was conducted is given in chapter 4. Chapter 5 lists all obtained results. In chapter 6 we draw our conclusions and in chapter 7 we discuss the implications of these findings for the course of object oriented programming. At the end of the thesis is a list of appendices, including the interview we conducted, the accompanying consent form, a rubric used to judge software quality, and one of the course's assignments, followed by its template and example solution.

Chapter 2

Background

2.1 Code Quality

There are many possible definitions for the notion of code quality. To avoid confusion, we will henceforth use the definition of Akour and Falah (2016): code quality or code readability is a metric defined by how easily source code can be understood, maintained, and adapted. We will make use of the rubric created by Stegeman, Barendsen, and Smetsers (2016) to judge the quality of submitted assignments. This rubric consists of ten generally agreed upon code quality aspects, and proposes a score distribution between one and four points for each of them.

2.2 Personal Prof

2.2.1 Background

Personal Prof is software written by doctoral candidate M. Klinik at the Radboud University. Its purpose is to give feedback on programming assignments written by students for the object-oriented programming course. The tool is written in Rascal (CWI Amsterdam, 2014), a metaprogramming language that provides a layer of abstraction over many different programming languages, allowing us to reason about code the same way in any of those languages, which makes porting Personal Prof to other languages for use in other courses easier. Personal Prof uses a rule-based classifying system to give feedback on general aspects like omitted access modifiers and broken naming conventions, as well as more specific details per assignment, such as the absence of certain classes or classes with unjustified dependencies.

2.2.2 Purpose

The goal of Personal Prof is to improve the feedback loop of the object oriented programming course, by reducing the amount of time between when

the students submit their work and when they receive feedback, and by filtering as many of the style-related issues as possible before the assignments get graded by a teaching assistant. This saves the assistants time correcting small, unnecessary mistakes and allows them to focus more on the functionality of the code, hopefully resulting in better feedback. Moreover, this more direct approach of providing feedback might improve the quality of the code written by students, both for the assignments and for future projects.

2.3 Related Work

Similar experiments, where tools are used to indicate software quality issues in code written by students, are described below. The results of these experiments are very different, and are highly dependent on the circumstances. What separates this research from the others is the fact that Personal Prof was specifically designed for the assignments of the object-oriented programming course. The specific feedback rules for each assignment should make the feedback more relevant and the tool more interesting to use. Moreover, the assignments are graded after the students hand them in, which should persuade the students to solve any issues they come across before that happens.

2.3.1 Negative Outcome

Keuning, Heeren, and Jeuring (2017) analyzed a large number code fragments from students collected by the BlueJ IDE (King's College London, 2009). To determine the quality of these samples, they also made use of the rubric created by Stegeman et al. (2016). Namely, for some of the quality aspects listed in the rubric, they determined whether or not a code snippet fulfilled that standard using PMD. PMD is a static analysis tool that recognizes many bad practices using rules, not unlike Personal Prof's rulesets. One of these rules, for example, checks if a variable is declared prematurely, which can cause confusion as demonstrated by Sasaki, Higo, and Kusumoto (2013), while another rule checks if a module contains too many methods. They also checked what happened to the code fragments over time to see if the issues get solved. Moreover, they analysed the effect certain BlueJ extensions had on solving them, most notably Checkstyle and PMD itself. They found that students create a large number of code quality issues, and that the usage of tools had little effect on how often they fix them. This is especially concerning considering that the PMD extension should tell them exactly what they have to change to be rated with high-quality. This is relevant to our research, as it shows that access to an automatic style-feedback tool like Personal Prof might not be enough to improve the quality of code written by students. For this reason, this thesis will also focus on the feedback on students' assignments and the assignment descriptions, to make sure

the students realize the importance of writing high-quality code. Moreover, Personal Prof also gives feedback specific to each of the assignments, which may be more relevant and may therefore make Personal Prof more alluring to use than the static analysis tools used by Keuning et al. (2017). Finally, this research uses a different metric to determine code quality than the tool that also helps students with their code quality, in order to measure the effect more objectively.

2.3.2 Positive Outcome

The research conducted by Cardell-Oliver et al. (2010) focuses only partly on software readability as defined above. However, that part is relevant to our thesis. In their research, they allowed students to use the Eclipse IDE (Eclipse Foundation, 2001) with Checkstyle. Much like Keuning et al. (2017), they also used Checkstyle to later determine the effectiveness of these tools. Unlike Keuning et al. (2017), however, the results of this research were more promising, showing significantly improved submitted programming assignments. This difference can be explained by the fact that the BlueJ code snapshots are not necessarily from people following courses on programming, while the students of this research received a lecture, tutorial, and laboratory session each week, just like the students at the Radboud University. One notable difference between this research and the research of Cardell-Oliver et al. (2010), is that the tools their students used can automatically indicate and resolve the quality issues they find, even without the programmer noticing. While their results are quite positive, we cannot guarantee that the improved code quality of the submitted assignments is due to a better understanding of software quality, or merely because the tools filtered most issues. Personal Prof's feedback is therefore uploaded after the submission, as it stimulates the students to think about the quality of their code beforehand, and eliminates the possibility that the tool resolved the issues automatically.

2.4 Interview

The main source of data for this research is collected through an interview. There are three main categories of interviews: structured, semi-structured, and unstructured. Structured interviews consist of a list of predefined questions and generally do not sway from their original order or content, while unstructured interviews do not have such a list, but usually ask relevant questions on the fly. Semi-structured interviews are somewhere in the middle, which is what we will be using for this research. We prepared a list of questions, but in cases where students mention important or ambiguous details, we should be able to ask follow-up questions. This way we can gain a deeper understanding of their opinions on code quality, on the feedback

they received, and on the tool itself. The interview was composed as described by Oates (2006) regarding content validity, construct validity and reliability.

2.5 Analysis

For this research, we will be performing qualitative analysis on the data generated from the interviews. This will be done through the use of codes (not to be confused with software code) that describe one particular type of event, such as a specific type of warning from Personal Prof, or a problem the students had when making the assignment. Appropriate fragments of the data are then marked with these codes to easily find each occurrence of said events. Codes can be defined in two main ways: inductively, where the statements from the students are turned into codes when they are encountered, or deductively, where codes from an existing framework are used where they apply in the data. Inductively defined codes are continuously updated as the researcher's understanding of the data improves. For example, in some cases two codes turn out to be so similar, that it might be better to join them, or in some cases one code might be overgeneralized and should instead be split into two different codes. This process makes sure the code framework always represents the data as well as possible, given the knowledge at that time.

To simplify the process of performing the qualitative analysis, we will make use of ATLAS.ti (ATLAS.ti Scientific Software Development GmbH, 1993). ATLAS.ti is software that helps perform qualitative research or analysis by providing tools that simplify finding complex patterns in unstructured data. In particular, it simplifies the process of marking fragments of audio files with codes, splitting and combining codes, and finding all occurrences of a code, among others.

Chapter 3

Goal

3.1 Research Question

The research question we answer in this thesis is:

How useful is automatic, style-related feedback to students of object-oriented programming?

As this question is difficult to answer directly, we will be looking at the accuracy of the tool, the students' usage of the feedback, and their opinion on its usefulness. Combined, these points give an indication of the usefulness of the tool in general. We translated these aspects into the following subquestions:

3.2 Research Subquestions

- *How well does the feedback tool recognize quality flaws?*
- *What do the students do with the generated feedback?*
- *How do the students value the automated feedback?*

Chapter 4

Methodology

4.1 Initialization

We started by enabling all students to access the tool as follows: when handing in the assignment of that week, the code is automatically checked by Personal Prof, which sends its feedback after about five minutes. After reading the feedback, the students are allowed to hand in a new version, which Personal Prof can correct again, etcetera. When the students are done, their code is checked by a teaching assistant for the final grade.

4.2 Data Collection

To determine the effect of Personal Prof, we conducted eight interviews: two for the first iteration and three for the second and third. The first iteration had one fewer because finding participants proved to be a challenge during the start of the coronavirus pandemic. We did the interview with every volunteering student soon after they received the feedback for the corresponding week. Along with the interview, we graded their assignment of that week using the rubric created by Stegeman et al. (2016), which can be found in appendix C, to get an idea of their level of writing high-quality code at that time.

4.2.1 Choice of Questions

The interview was split up into three main sections: one about the students' views on code quality, one about the performance of Personal Prof and one about the feedback of the course in general.

Students' views on code quality

In this section, the students were first asked to name the three most important aspects of code quality in their opinion. We deemed this information

important, as it is hard to write good code without knowing what conditions it should satisfy in order to be considered high-quality.

The second question of this section asks the students to motivate their choice from the previous question. It was important to verify that the students were not simply regurgitating memorized code quality aspects, but were really motivated to write good quality code themselves.

Personal Prof 's performance

If the interviewees used Personal Prof to get feedback on their latest assignment, we asked them what feedback the tool gave them and what changes they made in their code as a result. We verified their answers by running the tool over their code and analysing the output.

The students were then asked whether Personal Prof gave them unclear feedback or feedback that they did not agree with, to see if the issues that Personal Prof does pick up on are handled well. The previous question verifies the completeness of the tool, i.e. does the tool find all issues in the code? This question verifies its soundness, i.e. are all found issues actually issues?

Feedback effectiveness

In the last section, we asked the students which parts of all feedback they received, both from Personal Prof and from the student assistants, were particularly useful to them. Moreover, we asked them for suggestions on how to improve the overall feedback loop of the course.

Quality Aspects

When grading the assignments, we mainly focused on the 'names', 'idiom' and 'decomposition' rows of the rubric by Stegeman et al. (2016), because either the tool did not have builtin features to check the other rows or because those rows were not applicable to these assignments. However, the rows 'comments', 'formatting' and 'expressions' were also kept in mind as such checks were possible to implement if necessary.

4.2.2 Choice of Assignments

We chose to analyze the 6^{th}, 9^{th} and 11^{th} assignment. Data collection didn't start until the assignment of week 6 about recursive data types, as it was the earliest assignment containing a sufficient amount of object oriented code to grade the quality of and because this allowed us to ask questions about earlier assignments in case Personal Prof gave little feedback for this one. The assignment of week 9 was about JavaFX and, since many programmers

consider it best practice to separate the user interface from the business logic (Akour & Falah, 2016), this week was a good opportunity to test whether the tool could pick up on those principles. Finally, week 11 was about object oriented design patterns, which are an important aspect of creating maintainable and high-quality code in general, and therefore a good indicator to see how the tool performs. Another plus for these specific assignments was that there was a two or three week break between each of them, which gave us time to adapt the tool where needed.

4.2.3 Participants

Originally, we planned to find interviewees by attending the tutorial lectures and picking students at random to ask whether they would like to participate. However, now that the university was closed, we had to use another method. We then chose to find participants using convenience sampling: we made an announcement that asked the students whether they would like to help improve Personal Prof and the feedback for object orientation in general. It received little attention, so after several attempts we had to bribe students with a free snack, after which we were able to find enough volunteers. We attempted to include both people who were quite adept and people who had trouble with the assignments in each iteration, so that we could try to improve the experience for each of them. Moreover, all of the participants had made use of Personal Prof for the assignment they were interviewed for. Table 4.1 lists for each of the participants what they study and what their average grade for the assignments was. While these grades appear to be quite high, the assignments are graded with a 0 if they were not made seriously, a 1 if they had serious problems, 2 if there were a few small issues and a 3 otherwise. While these students certainly knew well what they were doing, this relatively simple grading scheme quickly results in relatively high grades, and their grades are quite representative of the entire group.

4.2.4 Interviewing

We conducted each interview as soon as possible after the student received all feedback on that assignment, both from Personal Prof and the grading teaching assistant. However, due to the university being closed, the first iteration had a slow start, resulting in a delay of one week between the feedback and the interview. Moreover, the interviews were held using either Skype or Discord, whichever the interviewee preferred. After having them sign the consent form found in appendix B digitally, we started the semi-structured interview found in appendix A.

	Study	Average Assignment Grade
1	CS	79%
2	AI	70%
3	AI	85%
4	CS	70%
5	AI	85%
6	AI	88%
7	CS	91%
8	CS	70%

Table 4.1: The average assignment grade of each of the participants, which is representative of the entire group (AI: Artificial Intelligence, CS: Computing Science).

4.3 Data Analysis

After each iteration, we analysed the data to improve the codes of our qualitative analysis. This research relies on a combination of inductive and deductive codes. A large number of codes related to code quality was deducted from the rubric by Stegeman et al. (2016), each row contributing between one and three codes as some categories grouped multiple events, while some students also brought up quality aspects that were not present in the rubric, resulting in inductively defined codes. The majority of codes for this research is inductive, as the feedback given by the students is often specific to the circumstances around the Radboud University, which makes most existing frameworks inadequate for this situation.

4.3.1 Interview

Each of the questions from the interview was used for a specific reason, as described below. The first section of the interview determines whether students have both the knowledge and motivation to write high-quality code, the second section checks Personal Prof's capability to aid them in this process, and the last section gauges how satisfied the students were with the feedback loop in general, which gives us more information about other aspects that might influence the final results. The rubric was used as an independent opinion on the quality of the code written by the students.

Students' views on code quality

The number of named quality aspects from the rubric served as an indication of how well the students know what quality guidelines to follow. If this

number was too low, we would consider adaptations to Personal Prof that should help the students understand these quality aspects better. Otherwise, this data was used to verify that the tool works as expected. The reasons the students mentioned for their quality aspects were mainly used for discussion. Most reasons can be considered fine, as long as the students mention it out of their own free will, and not because they learned it was important.

Personal Prof's performance

The results of this section of the interview were used to remedy any problems the students had with the tool, such as flawed rulesets or unclear feedback descriptions, and to find possible improvements for further iterations. They were also used to determine how content the students were with the software.

Feedback effectiveness

This final section was used to determine whether there were other factors that might have had impact on the way students think about software quality. Other issues regarding software quality that might have obstructed the students were also addresses with this data.

4.3.2 Rubric

We used these results to determine how well Personal Prof can detect quality issues in source code. If Personal Prof does not find any issues with some code while the rubric does, we could add functionality to the tool in order to improve the students' ability to adhere to those aspects as well. Otherwise, if Personal Prof and the rubric are both positive about a program, it serves as confirmation that Personal Prof is working as intended. All assignments were graded by the same person in order to minimize the effects of personal preference.

4.3.3 Adaptation Phases

After analysing the data each iteration, we used the break to adapt the tool in an attempt to resolve any issues the students might have had while using it, and to improve its performance for the next iteration.

4.4 Combining Results

After having analysed the data of the last iteration, we combined and compared portions of the results that were relevant to answering the research questions, in order to clarify our conclusions.

Chapter 5

Results

5.1 Summary

This section contains a summary of all data collected in this research. For a detailed analysis for each iteration, please refer to the next three sections.

5.1.1 Codes

5.1 lists all of the codes used in this research, along with how many of the students mentioned it each iteration. The codes under 'Quality aspects from rubric' and 'Feedback on the assignment' were deducted from the rubric of Stegeman et al. (2016) and Personal Prof's source code, respectively. The other codes were all inductively defined according to the responses of the participants during the interview.

Code	**1**	**2**	**3**
Quality			
Quality aspects from rubric			
(Comments) there should be clear and concise comments explaining parts of the code	1	1	0
(Decomposition) each function/class has one clear task only	0	2	0
(Flow) there is little to no duplicate code	1	2	1
(Formatting) the code should be indented properly and easy to read	1	0	1
(Headers) the code should have clear and accurate documentation	1	0	1

(Layout) the positioning of elements within the source code is logical and easy to follow	1	0	0
(Modularization) the code is separated into logical modules, with limited communication between them	0	1	1
(Names) functions and classes should have clear and concise names	1	2	3
Quality aspects from students			
Functions should be relatively short	0	3	1
Lines should be relatively short	0	1	0
The code should be abstract	0	1	0
The code should run efficiently	1	0	1
The code should work as expected	1	0	0
Reasons for quality aspects			
The student prefers more concise code	0	0	1
The student has trouble finding variables sometimes	0	0	1
The student learned that it is important	0	2	0
The student argues that it makes debugging easier	0	0	1
The student argues it makes the code more flexible	0	1	0
The student argues that it makes the code easier to understand for other people	2	3	1
The student considers it easier to get back to after a break	2	1	2
The student argues that it can be read more quickly	0	1	1
Personal Prof			
Feedback on the assignment			
Class A should be the only one to use B	0	1	0
Errors found, did not check	0	0	1
Issue with access modifiers	2	0	2
Issue with naming convention	1	0	0

Issue with handing in	0	0	1
Tests are missing or failing	1	0	0
Redundant class	1	0	0
Class missing	0	0	1
No issues found	2	3	1
Student's reaction on the feedback			
The student fixed larger issues, but kept smaller ones	0	0	1
The student chose not to adapt the hand-in	0	0	1
The student kept adapting the code until Personal Prof approved	2	3	1
Complaints			
The student was sometimes unable to find the feedback	1	0	0
Personal Prof takes too long to check the assignment	0	1	0
Personal Prof complained about code only used for testing	1	0	0
Personal Prof didn't check the code because there was a compile time error	0	0	1
Personal Prof enforced overriding all inherited methods	0	0	1
Personal Prof should perform more checks	0	1	0
Personal Prof sometimes takes much longer to provide feedback	0	1	0
Compliments			
It is clear how to use Personal Prof	0	1	0
It is easy to follow the feedback from Personal Prof	0	2	0
It is nice to have the assignment pre-checked before you get graded	0	2	0
Personal Prof can warn you if there is a large part of the assignment missing	0	0	1
Personal Prof helps fixing small issues	2	3	2

Feedback on the course in general			
Points of improvement			
An example solution would help	1	0	0
Assistants disagree on how the code should be	1	1	0
The student was unaware that code should be split into separate files	0	0	1
Personal Prof mentions missing classes that were not specified in the assignment	0	0	1
Personal Prof gave positive feedback while the assistant did not	1	1	0
The assignment description is unclear	1	0	0
The assignment is too strict about how to solve a problem	0	0	1
The feedback of the assistant is too brief	0	1	3
The lectures and tutorial were out of sync	0	0	1
The lectures were not recorded	0	0	1
The slides contain a lot of code and little explanation	0	0	1
When receiving negative feedback, it is unclear how it should have been done instead	1	1	0
Compliments			
Even when the assignment was well done, the assistants still give points of improvement	0	0	2
Snake assignment			
The student had difficulty with the observer pattern	-	2	-
The assignment was fun	-	1	-
The assignment description was vague	-	1	-
The template was very large	-	1	-

Table 5.1: All codes from the analysis, along with how many students mentioned it in each of the three iterations.

5.1.2 Grades

The style grades according to the rubric of Stegeman et al. (2016) for each student can be found in table 5.2. Since each of the six categories gets a minimum of one point, we calculated the grades by subtracting six from the achieved score and dividing that by eighteen rather than twenty-four.

	Score	Style Grade	Average Assignment Grade
1	21/24	83%	79%
2	22/24	89%	70%
3	21/24	83%	85%
4	21/24	83%	70%
5	20/24	78%	85%
6	21/24	83%	88%
7	20/24	78%	91%
8	21/24	83%	70%

Table 5.2: Graded work of all iterations of students, according to the rubric.

5.2 First Iteration

After performing the qualitative analysis, we ended up with a list of categories, all of which are split up into subcategories of codes, which we will describe in detail below. Each of the codes has a short explanation and/or one or two quotes to illustrate why it was added to the list of codes. Moreover, for each of the codes is noted how many students mentioned it applied to them.

5.2.1 Quality

Quality aspects from rubric

Note that these codes were deducted from the rubric from Stegeman et al. (2016) and that only codes that applied to at least one of the students are included. For a better understanding of these codes, please refer to the rubric itself in appendix C.

- (Comments) there should be clear and concise comments explaining parts of the code

  Every function or every few lines of code should have at least one comment explaining the effect it has on the state or parameters.

> "I just think it's very important that every part of the program has comments that explain what that part does, because that makes it easier to find things." - Student 1

- (Flow) there is little to no duplicate code

  All code segments should be reused where possible. Duplicate code segments can be confusing as changing just one of the duplicate instances does not change the program in the way it is expected to, because in some cases it will use the other instance(s) and reproduce the old behavior instead.

  > "The code should contain little duplication (. . .) because it's hard for other people to work with if they have to change something" - Student 2

- (Formatting) the code should be indented properly and easy to read

  The code should adhere to the indentation conventions of the language it is written in, and use blank lines where needed to improve the readability.

  > "It should be well-formatted and readable, as that allows other people to read your code more quickly." - Student 1

- (Headers) the code should have clear and accurate documentation

  Each module of the code has clear documentation on how to use the parts related to that module.

  > "I would say the documentation, so that it's clear what the code does. So if you look at the code after half a year, you can quickly see what the code does, and others can also quickly see what the code does." - Student 1

- (Layout) the positioning of elements within the source code is logical and easy to follow

  The positioning of methods relative to each other, and of expressions within methods, is easy to follow. Related methods should be grouped, such as Java's getters and setters, while constructors and other special methods should have consistent locations throughout the program.

  > "It should be structured, so if you look at it after, say, a year, or when someone else looks at your code, that you don't have to think very long about where to find things." - Student 2

- (Names) functions and classes should have clear and concise names

  Clear method names can increase productivity, as searching for the right methods is easier when the method has an obvious name. Moreover, a clear name can diminish the need for documentation (Akour & Falah, 2016).

  > "And for clear function names, it just makes it easier to understand what the code does, so learning how it works can be done more quickly." - Student 2

Quality aspects from students

Some students named aspects of code quality that didn't fit any of the rubric's categories. Those aspects are listed here.

- The code should work as expected

  One of the students argued that the best aspect of software quality is that the software works.

  > "It's most important that it works, because that's the reason you do it at all." - Student 1

- The code should run efficiently

  One of the students mentioned that code that runs more efficiently can be considered of higher quality, even if efficiency comes at a cost of more complex code to a certain extent.

  > "If it spends an hour to calculate one plus one, then it's not of much use either." - Student 1

Reasons for quality aspects

The following is a list of codes about the main reasons students thought the aforementioned aspects were important. In this case, both of these codes are linked to all of the mentioned quality aspects from the rubric, including duplicate code. The students associate high quality with code that is easy to use for people who are not familiar with it, whether those are people new to the code in general, or themselves after a break.

- The student argues that it makes the code easier to understand for other people

  > "It's useful that if you look at the code again after half a year, you can quickly see what it does and that someone else can also quickly see what it does." - Student 1

- The student considers it easier to get back to after a break

  > "It should just be clear for yourself and others, so if you look at the code the next week or after a long break, you won't have to think long about what it means." - Student 2

Note that the students do not mention an increased risk of flaws in lower-quality software, even though this is apparent in case of duplicate code segments.

5.2.2 Personal Prof

Feedback on the assignment

These codes describe the feedback Personal Prof gave the students for the corresponding assignment. Note that these codes were deducted from Personal Prof itself.

- Issue with access modifiers

  Either some fields or methods have missing access modifiers (public, protected or private) or use them in a way that may cause problems, such as non-final attributes being public.

  > "We wrote one of our static constants in uppercase letters, but without final modifier, even though it should have been final, so that was useful feedback." - Student 1

- Issue with naming convention

  For Java, almost everything is written in `camelCase` by default, except for class names, which also start with an uppercase letter, and constants, which are written in `SCREAMING_SNAKE_CASE`. Breaking this convention will trigger this warning.

  > "That a static final constant should be written in uppercase characters." - Student 2

- Tests are missing or failing

  This code combines warnings having to do with test cases: either there aren't enough test cases defined, or not all of them are passing.

  > "One of our tests failed, but that was kind of intentional." - Student 2

- Redundant class

  This warning is triggered when the student created a class that is not needed to complete a specific assignment. It may be too strict, as it is usually used to prevent the student from making any classes other than what the assignment tells them to.

  > "I created an inner class that extends comparator to sort the list. This wasn't strictly necessary, but it was useful for testing, as it's easier to see if two sorted lists contain the same elements. However, Personal Prof said I wasn't allowed to make that class." - Student 1

- No issues found

  If Personal Prof cannot find any issues with the code, this is the output it gives. All of the students interviewed received this feedback eventually.

Student's reaction on the feedback

This category contains codes having to do with what the students choose to do with Personal Prof's feedback, i.e. do they improve their code, ignore the feedback, or do something in between?

- The student kept adapting the code until Personal Prof approved

  So far, all of the students reacted to the feedback in the same way: they kept adapting the code until Personal Prof said it looked good.

Complaints

This category hosts codes having to do with complaints the students had about Personal Prof.

- The student was sometimes unable to find the feedback

  Though student 2 should have been able to see the feedback from Personal Prof, the location might be unclear.

  > "Yes, I used Personal Prof, but the problem is that I often couldn't find the feedback. But the person I work with was always able to find it." - Student 2

- Personal Prof complained about code only used for testing

  Personal Prof may have been too strict on the aforementioned redundant comparator class that student 1 used for testing. On the other hand, the class that was only used for testing should not have been included in the program itself.

> "So Personal Prof said I wasn't allowed to use the inner comparator class, but the reason I disagree with that is because we learned about inner classes this week, so apparently the inner class wasn't bad after all." - Student 1

Compliments

Fortunately, the students were quite enthusiastic about Personal Prof as well, as can be seen from the following code:

- Personal Prof helps fixing small issues

  The students were both quite content about Personal Prof helping them fix smaller issues (mainly access modifiers).

  > "I kept forgetting to use public or private, so it was very nice that Personal Prof reminds you of these things (...) it's just really nice, even though they're just small things, they're usually easy to correct and the TA won't have to waste time on them after that." - Student 2

5.2.3 Feedback on the course in general

Points of improvement

Finally, this category accumulates all feedback the students gave on the feedback loop of Object Orientation in general.

- An example solution would help

  Example solutions are usually never published because students tend to put them online, which gives problems when the teacher wants to reuse old assignments. However, it is important that the students know how to improve their code after they receive feedback.

  > "How someone would go about implementing the so-called perfect solution, like an example solution or something, would really help me understand how to improve on the things I did wrong, and I think it would help others who are having trouble as well." - Student 2

- Assistants disagree on how the code should be

  What qualifies as high quality code is a relatively subjective manner. This also applies to the teaching assistants as can be seen from this example:

> "We were making an assignment during the tutorial and got help from one of the teaching assistants, who eventually told us our solution was very good, while another TA, who graded the assignment, deducted points because they thought the implementation wasn't that good. The tastes differ quite a bit apparently, which I think is weird." - Student 2

- Personal Prof gave positive feedback while the assistant did not

  The feedback from Personal Prof is meant to help students with smaller, mostly style-related issues and cannot replace feedback from the teaching assistants. This may be somewhat unclear.

  > "Sometimes you can see that Personal Prof does give feedback on certain issues but the TA doesn't, or the other way around." - Student 1

- The assignment description is unclear

  Sometimes the assignment description is interpreted in a different way than intended.

  > "The biggest annoyance I have doesn't really have to do with the student assistants or Personal Prof, but sometimes the requirements of an assignment turned out to be different than what I understood from the assignment description" - Student 1

- When receiving negative feedback, it is unclear how it should have been done instead

  This code is linked to the example solution, which the student proposed to solve this issue.

  > "As a point of improvement, I'd say that it's difficult to know how to improve the code after getting negative feedback. We quickly hear things that could have been done better, but *how* to improve it is often so ambiguous or general that I just don't know how to go about implementing it. I do understand the feedback, I just don't know how to improve my work afterwards." - Student 2

5.2.4 Rubric Grades

We graded the style of each of the students' assignments with the rubric. For this relatively small assignment, there were very few issues, though some might argue that comments could be used more generously. The resulting grades can be found in table 5.3.

	Names	Idiom	Decomposition	Comments	Format	Expression
1	4/4	3/4	4/4	2/4	4/4	4/4
2	4/4	4/4	4/4	2/4	4/4	4/4

Table 5.3: Graded work of the first iteration of students, according to the rubric.

5.2.5 Personal Prof Adaptations

UI separation

For the next iteration, all students made two assignments related to JavaFX. Assigment 9 is about programming a basic snake game with a graphical user interface. The template the students first started with mixed the user interface and input handlers with the game logic. For a small project as this, doing so is fine, but most purists will argue it is bad for code quality in larger projects (Akour & Falah, 2016). However, it can be difficult to come up with an architecture that separates these nicely, especially for students that have not been programming for long. We decided to adapt the assignment description and starting template and added a specific ruleset to Personal Prof for assignment 8 that helps the students achieve an architecture where the game logic and graphics are separated. The assignment description can be found in appendix D, while the starting template and example solution can be found in appendices E and F, respectively. Credit goes to the course organizers for providing the initial template.

Assignments 8 and 9 were the only assignments Personal Prof did not have rulesets for, so adding a few was necessary for this research. The reason we strayed further from our original plan by not only changing Personal Prof but also changing the assignment is because we consider this a nice test of how Personal Prof can guide the students as well as a good opportunity for the students to learn different architecture design.

Completion check

In response to the feedback from this iteration, we also added a list of key functionality points that the program should contain to the assignment. If an assignment checks all those points, it can be considered done. This way, we hope the goal of the assignment is more clear to the students.

5.3 Second Iteration

Like in the previous iteration, here is a categorized list of all codes that are relevant to the research so far, followed by the adaptations to Personal Prof

that were inspired by the feedback. The results from the previous iteration are excluded. The biggest differences of this iteration are summarized here.

Compared to the first iteration, none of the participants mentioned formatting, documentation or the positioning of elements within the source code as important aspects of code quality in this iteration, but unlike previous iterations, aspects like decomposition and modularization were mentioned this time. For the quality aspects not included in the rubric, the students from this iteration did not mention efficiency nor correctness, but they did value abstraction and shorter functions and lines. Their reasoning behind certain code quality aspects was broader than just making the code easier to understand, as improved flexibility was also named this iteration, which was one of the reasons to adapt this iteration's assignment.

Issues with access modifiers, naming conventions or missing classes were not present in this iteration, rather there was one case where Personal Prof mentioned that a certain class should not use a specific dependency, which was unjustified and fixed later.

Some of the students of this iteration had issues with how long Personal Prof took to grade the assignments. On the other hand, they also mentioned that they were pleased with the fact that they could have their assignment checked before it is graded, and how easy it is to use Personal Prof.

None of the participants mentioned that an example solution was necessary or that the assignment description was unclear, points of focus that were addressed in the previous iteration by changing the snake assignment. Rather, some of the students pointed out that the feedback they got from the teaching assistant was quite brief.

Finally, we decided to ask each of the students from this iteration to give feedback on the snake assignment, to see if they encountered problems making it. Many students considered the assignment difficult, mostly due to the observer pattern. These codes can be found near the end of this section.

Like in the previous iteration, the style grades of the participants from this iteration can be found in table 5.4 at the end of this section. Note that the starting template largely determined most of these aspects, as the assignment only required students to implement specific parts of the program.

5.3.1 Quality

Quality aspects from rubric

- (Comments) there should be clear and concise comments explaining parts of the code

> "And if a code segment does more than one thing, you should use comments to describe what's happening." - Student 5

- (Decomposition) each function/class has one clear task only

  Two of the students actively tried to keep their methods focused on one task, rather than having one large method that does multiple things.

  > "I usually split large functions up into different smaller ones that perform one operation each, to make it a bit more structured for myself." - Student 4

  > "Functions shouldn't be too long, they should be split up into smaller functions that do one thing each." - Student 5

- (Flow) there is little to no duplicate code

  > "The code should contain little replication, as those can cause errors when you change one part but forget to change the replica as well." - Student 3

- (Modularization) the code is separated into logical modules, with limited communication between them

  One of the goals of adapting the snake assignment was to teach students that grouping related parts of the code, such as functionality around libraries, into modules and minimizing their use in other modules can improve the quality of the software. One of the three students mentioned this during the interview.

  > "Keeping similar things in separate modules, like the graphics from snake, but maybe it can be done for other things as well." - Student 3

- (Names) functions and classes should have clear and concise names

  > "I think the names are most important: they allow you to know what a function does even if the function itself is unclear." - Student 5

Quality aspects from students

- Functions should be relatively short

  All students mentioned that they try to keep their methods short. This is similar to the decomposition code from the previous category, but there is a difference keeping functions short and making sure each function is responsible for exactly one action, hence the separate code.

  > "In the beginning, there was a large emphasis on keeping functions short." - Student 3

- Lines should be relatively short

  Long lines can cause trouble reading, which can in turn make the code harder to understand.

  > "If you write a line of code, make sure that you don't move past the line NetBeans indicates as the maximum line length." - Student 5

- The code should be abstract

  While most professional software developers will argue that premature optimization or abstraction is a bad thing for productivity, abstraction can improve the readability of code significantly and can lower the probability of encountering issues (Sadou, Tamzalit, & Oussalah, 2005).

  > "During the OOP sessions, they really try to push 'abstraction, abstraction, abstraction,' so that's important I guess." - Student 4

Reasons for quality aspects

- The student learned that it is important

  Though undesirable, some of the quality aspects mentioned by students were only mentioned because they were taught it was important.

  > "I mentioned separating concerns because I learned it is important, I don't completely understand why." - Student 3

- The student argues it makes the code more flexible

  > "Replication because otherwise you can't really update it, if you want to change it, you have to change it everywhere and that's a bit chaotic." - Student 3

- The student argues that it makes the code easier to understand for other people

  > "It should be clear what it does so other people can look into it as well." - Student 3

- The student considers it easier to get back to after a break

  > "If you make extra functions so each of them is shorter, it makes changing the code much easier for yourself." - Student 4

- The student argues that it can be read more quickly

  One student also considered how much time can be saved by keeping functions clear and lines short.

  > "If everything is nice and compact, you can see 'oh, this function does that and this function does that' much more quickly." - Student 5

5.3.2 Personal Prof

Feedback on the assignment

- Class A should be the only one to use B

  The new ruleset for the snake assignment disallowed the use of an inner class instead of a lambda expression. Since the Java documentation does not comment on which is the preferred way of doing it, this was later changed to allow for inner classes as well.

  > "Using an inner class rather than a lambda expression to add input listeners would result in Personal Prof saying those inner classes are not allowed to depend on listeners." - Student 4

- No issues found

Student's reaction on the feedback

- The student kept adapting the code until Personal Prof approved

  Like in the previous iteration, all participants kept improving the assignment they handed in until Personal Prof said it looked good.

Complaints

- Personal Prof takes too long to check the assignment

  While not giving students access to Personal Prof directly but rather through a five minute delay was an intended design choice, it does have its side-effects.

  > "A tiny point of improvement would be that, because it takes a few minutes before Personal Prof gives its feedback, you already think to yourself 'ah, finally done programming' and immediately close your laptop. Then after an hour you realize you forgot about the feedback." - Student 4

- Personal Prof should perform more checks

  One teaching assistant deducted points from this student's assignment because they prefixed interface methods with public, which is redundant according to the Java documentation (Gosling, Joy, Steeleand G. Bracha, & Buckley, 2015). She justly mentions that Personal Prof should have noticed that, but in reality, Personal Prof would have advised against omitting the access modifiers, as it is programmed to check whether they are always explicit. We adapter Personal Prof for this reason.

  > "We used public methods in an interface, and the TA said that wasn't allowed, which is something Personal Prof should have mentioned in my opinion." - Student 3

- Personal Prof sometimes takes much longer to provide feedback

  For one of the pairs, how long it takes for Personal Prof to upload the feedback appeared to be inconsistent, and sometimes it took much longer than it should.

  > "Sometimes we got the feedback very quickly and sometimes it took really long. It seemed to be quite inconsistent. Eventually we just stopped waiting." - Student 5

Compliments

- It is clear how to use Personal Prof

  The choice to embed Personal Prof into BrightSpace can be convenient.

  > "It's just very clear, you just have to upload your assignment as usual, and you automatically get the feedback after about five minutes." - Student 4

- It is easy to follow the feedback from Personal Prof

  The students indicate that Personal Prof mostly giving feedback on small issues they can correct quickly is a good thing.

  > "The feedback you get is usually not feedback you have to think about very long, it's obvious what you should change." - Student 4

- It is nice to have the assignment pre-checked before you get graded

  While Personal Prof provides very little correctness or completeness checks, other than running tests and asserting the existence of certain classes, these students are more confident about their code after

receiving positive feedback from Personal Prof. Like the code about Personal Prof being positive while the teaching assistant was not, this code should be kept in mind as Personal Prof cannot fully check the assignment requirements, so this sense of reassurance may be unjustified.

> "I just liked the feeling of having your assignment checked before you actually have your assignment checked, it feels reassuring." - Student 4

> "Because this course is more difficult than Programming 1 or 2, I like the fact that you get a small check-up. When I get positive feedback from Personal Prof, I know that I probably got at least the basics of the assignment." - Student 5

- Personal Prof helps fixing small issues

> "What I like is that it corrects small mistakes that you can correct easily, which can improve your grade, even though you still know you understood it correctly but just needed to pay closer attention. These corrections quickly teach you not to make these kinds of mistakes." - Student 4

5.3.3 Feedback on the course in general

Points of improvement

- Assistants disagree on how the code should be

> "We often get very different feedback, probably from different TAs because they think very differently about our code." - Student 5

- Personal Prof gave positive feedback while the assistant did not

> "It did happen that Personal Prof said the assignment was well done but the TA almost gave us an insufficient." - Student 3

- The feedback of the assistant is too brief

  While hard to do for a course with over 600 participants, some students might profit from more elaborate feedback.

> "For Programming 1 and 2 we got more specific feedback with a long explanation what could be improved, which was

> nice, while here it's usually just a short description of what's wrong, without explanation on how to improve it." - Student 3

- When receiving negative feedback, it is unclear how it should have been done instead

  > "For that feedback, I need an explanation on how to improve the assignment." - Student 3

  > "I didn't always know how to improve the assignment, so I had to ask a TA or someone else first." - Student 5

Snake assignment

- The student had difficulty with the observer pattern

  The observer pattern gave many students a hard time, both because they were not familiar with it and because it was unclear to them why it was used in the first place. The assignment description was adapted to reflect on this.

  > "I didn't really understand 5.3, the part about the listener, it was a difficult section." - Student 3

  > "The concept of listeners was a bit vague for me in the beginning. It did improve a little, but I still don't understand them a hundred percent." - Student 4

- The assignment was fun

  Fortunately, some considered the assignment more fun than other assignments.

  > "Because the assignment was interactive, it was more fun to play around with than the other assignments." - Student 5

- The assignment description was vague

  This student had trouble with the assignment description in general. While a large part of the assignment is the same as it was before, the section on the observer pattern is completely new. We hope that the added explanation will prevent this issue in the future.

  > "I thought the assignment description was a bit vague, I had to look through the lecture and the tutorial thoroughly. Maybe it was the wording or a lack of details, but I didn't

> really understand what to do. But with some help, I was able to do it eventually, but there was a lot of searching involved, especially in the beginning." - Student 5

- The template was very large

  The assignment template is also largely the same as it used to be, what is present is relatively minimal code required for a snake game. While the large amount of template code could possibly be a problem, solving it is beyond the scope of this thesis.

  > "Of course, we started with quite a bit of code already, so inserting new functionality is quite difficult because you first need to understand what all the other code does." - Student 4

5.3.4 Rubric Grades

	Names	Idiom	Decomposition	Comments	Format	Expression
3	4/4	4/4	4/4	1/4	4/4	4/4
4	4/4	4/4	4/4	1/4	4/4	4/4
5	4/4	4/4	4/4	1/4	3/4	4/4

Table 5.4: Graded work of the second iteration of students, according to the rubric.

5.3.5 Personal Prof Adaptations

Inner classes

The volunteers provided plenty of feedback on the adapted Snake assignment. One point of improvement was that the mouse and keyboard handlers in `InputHandler` didn't necessarily have to be added through a lambda expression; an inner class would also be acceptable and, in some cases, even preferable. However, our implementation of the Personal Prof ruleset would then complain that `InputHandler` should be the only class depending on JavaFX's input handler classes. We adapted the rule to check if `InputHandler` or any of its dependencies are the only classes depending on input handlers.

Access modifiers

Due to the substantial number of times the students received feedback having to do with access modifiers, we noticed that Personal Prof forced students

to also add public access modifiers to methods in an interface and private access modifiers to enum constructors, even though these are redundant and discouraged (Gosling et al., 2015). Moreover, at least one of the students we interviewed so far received a lower grade due to marking interface methods public while the TA argued doing so was bad. We decided to add exclusions to Personal Prof's current access modifier rules and added new warnings to notify students when their code contains redundant access modifiers. We also notified the students of this change in order to prevent confusion.

5.4 Third Iteration

Unlike the previous iterations, this final iteration does not have a section on Personal Prof adaptations, as any changes will not affect the remainder of this research. The results from this section will still be used to improve the feedback of the course, but it will not be logged here. A short summary of notable differences compared to the last iteration will follow.

The students from this final iteration named formatting and documentation as important aspects of code quality, but did not mention comments or decomposition, unlike previous iteration. Regarding quality aspects that are not present in the rubric, the students from this iteration mentioned efficiency as a good code quality metric.

There is a larger variety of motivations behind programming high-quality code than in the previous iterations. Among others, the students mentioned conciseness, trouble finding variables and ease of debugging as reasons to maintain code quality. Moreover, none of them said that they do it solely because they learned it was important.

In contrast to previous iterations, this iteration did have students that reacted to Personal Prof's feedback in a different way than fixing the issues, choosing not to implement some or all of Personal Prof's suggestions.

There were relatively few compliments for Personal Prof this iteration, but also few complaints.

The students of this iteration had quite a few points of improvement for the course in general. Most notably, all of the students mentioned that the feedback of the teaching assistant was too brief. The assignment being to strict about how to solve problems, Personal Prof expecting classes that were not mentioned in the assignment, and the slides from the lectures containing too much code and too little explanation were also mentioned. The students did note that they appreciate getting small points of improvement from the assistants, even if their assignment was well done.

The grades for the final iteration's assignment can be found in table 5.5. Like in the previous iterations, there were few issues with the code. The

lack of comments is not a big problem in this context as the purpose of the code for any specific assignment is already clear to both the students and the teaching assistants.

5.4.1 Quality

Quality aspects from rubric

- (Flow) there is little to no duplicate code

  > "I think it's also important that stuff isn't repetitive. So that you make functions that you can reuse and maybe alter in a slight you so that you don't have to write four classes for four things, but just one and then have it do different things depending on what you want. So don't repeat stuff." - Student 6

- (Formatting) the code should be indented properly and easy to read

  > "Of course formatting, because that makes it quicker to read." - Student 8

- (Headers) the code should have clear and accurate documentation

  > "For me personally, it's documentation, because making yourself documentation makes it way easier to debug your system in the end. If you always write down what a function does, then you can go through your code way faster." - Student 8

- (Modularization) the code is separated into logical modules, with limited communication between them

  > "I'm not sure about the name, but having private or public and that kind of stuff in front of every variable and function and so on. That way you cannot access things that you shouldn't have access to" - Student 6

- (Names) functions and classes should have clear and concise names

  > "Sometimes I cannot find variables because I gave them weird names, so maybe it's important to give variables good names." - Student 7

Quality aspects from students

- Functions should be relatively short

  > "If you have too much code somewhere, you should make a separate function to keep it clear, so that you can also understand what you did previously." - Student 7

- The code should run efficiently

  > "If you can do something to make the code run faster, then you should do so. Currently, we're learning about running two processes simultaneously, so something like that." - Student 7

Reasons for quality aspects

- The student argues that it makes the code easier to understand for other people

  > "I'd say just naming of variables and methods, so just giving them logical names that other people can understand and you can understand yourself if you look at your code two weeks later. Not just a, b or x, y or something." - Student 6

- The student argues that it makes debugging easier

  As mentioned before by student 8, proper documentation can make code easier to debug.

- The student considers it easier to get back to after a break

  As mentioned before by student 7, as a reason to keep functions short and readable.

- The student argues that it can be read more quickly

  > "I have it so often that it took me so long to read someone else's code, you know? You can always run through your own code very fast, but if you look at someone else's code it can take between minutes and hours depending on how you format and how you name the things." - Student 8

- The student prefers more concise code

  As mentioned before by student 6, keeping repetition to a minimum makes your code shorter.

- The student has trouble finding variables sometimes

  As mentioned before by student 7, a reason to give variables and functions proper names.

5.4.2 Personal Prof

Feedback on the assignment

- Errors found, did not check

  In some cases, it might be useful for Personal Prof to perform checks even if there are compile-time errors.

  > "There was a warm-up exercise where we had to turn a class into a class with generics. There was an error in the code, but a runtime error, and we had to change it so that it would be a compile-time error. That was the exercise. I did that and I uploaded it and the first feedback was that there were compile errors, and nothing else. So when there are compile-time errors apparently there is nothing else Personal Prof looks at, so we had to comment that part out and upload it again to get feedback." - Student 6

- Issue with access modifiers (public, protected, private)

  > "After reuploading, it told me that there were some access modifiers missing, and that I should remove some, because I had some in an interface and it told me that I should remove them." - Student 6

  > "It said something was redundant for an enum constructor." - Student 7

- Issue with handing in

  While this will not be a problem for the future, as the assignments from previous year will already have been checked, Personal Prof does not grade reused assignments.

  > "Because I handed in a text file with my name to reuse the assignment from last year, but I can live with that." - Student 8

- Class missing

  To clarify, their enum was called 'constant', without a capital letter, which is why Personal Prof did not find it.

> "And we got 'there should be an enum called Constant', but I thought we had that." - Student 7

- No issues found

Student's reaction on the feedback

- The student chose not to adapt the hand-in

  One of the students from this iteration mentioned ignoring the feedback if they think their solution is good or even better.

  > "With this feedback, we didn't do much, because we found out later that we did something differently with an enum than intended, but we thought the way we solved it was pretty good. We've done that before: if Personal Prof says something about it but we think 'our solution works as well and, in our eyes, looks better', then we usually don't do much with the feedback." - Student 7

- The student fixed larger issues, but kept smaller ones

  One of the students does fix the larger issues mentioned by Personal Prof, but might skip the smaller ones to save time.

  > "I look at it before the deadline and I look if I have such big mistakes, and then I look it up and look if I can do something better. Sometimes I'm skipping it, for things like 'it should be not all caps', you know? The problem is I have many subjects this quarter and I am a little bit overwhelmed so I don't have the ability to dig that deep into subjects. (...) I will not change it for this assignment, but for the future assignments, I'm a little bit smarter." - Student 8

- The student kept adapting the code until Personal Prof approved

Complaints

- Personal Prof didn't check the code because there was a compile time error

  > "The only thing that was a bit annoying is that it only told me this won't compile, and, I mean this wasn't in any of the other assignments and once you did this you could just do the other stuff, but the fact that the Personal Prof didn't check for anything else because there were compile time errors, that was a bit annoying, because I had to change it and then reupload it again." - Student 6

- Personal Prof enforced overriding all inherited methods

  This issue was fixed before the first iteration, but earlier Personal Prof enforced that subclasses override all methods from superclasses.

  > "I think it was fixed already, but we had an exercise with a bunch of abstract classes and subclasses, and there were like three methods in the superclass, and in the subclass we override one of them. Then the Personal Prof told us we actually had to implement all three, but the whole point of inheriting methods from classes is so that you don't implement them again, so that was a bit strange." - Student 6

Compliments

- Personal Prof can warn you if there is a large part of the assignment missing

  While it did not help the student in question in this case unfortunately, Personal Prof can warn you about missing parts of the assignment.

  > "The first assignment I made, I downloaded the template, I made it and handed it in. I was a bit confused from the feedback I got from Personal Prof, because it says 'you should use this class and this class', and I was like 'I did'. Then someone told me this Personal Prof is still learning, so I thought maybe it needs to learn, because it was the first assignment. But in the end we found out that I totally handed in the original template." - Student 8

- Personal Prof helps fixing small issues

  > "I think the Personal Prof is just useful because it quickly gives you very obvious things that you just forget. So if you have, like, ten classes and a bunch of methods, then you quickly forget that stuff. So it is pretty useful I would say, when it's done well." - Student 6

5.4.3 Feedback on the course in general

Points of improvement

- The feedback of the assistant is too brief

  > "Getting more feedback or more precise feedback would of course always be good, but I don't think that the TAs could

> go that deeply into everyone's code and suggest every little change they could make." - Student 6
>
> "Generally speaking, we didn't get very much feedback, usually just a few lines, which is a shame." - Student 7
>
> "I've only gotten feedback like 'great implementation', 'okay' or 'reuse', so it could be a little more elaborate." - Student 8

- The student was unaware that code should be split into separate files

  For the students unfamiliar with Java, the first lecture should mention that each class should preferably be in a separate file.

  > "For the first assignment we got a fail because we didn't know that we had to split our program into separate files, which was never mentioned in the lecture. Since we did everything before in c++, this came out of nowhere for us." - Student 7

- Personal Prof mentions missing classes that were not specified in the assignment

  The assignment description should always make it clear what Personal Prof expects from a solution, but in this case a clarification is needed.

  > "One time we heard that there had to be a specific interface, but we solved it differently, and the assignment didn't even specify that we had to use such an interface. And since we already did it our own way, we didn't change it." - Student 7

- The assignment is too strict about how to solve a problem

  Usually, the assignments propose one way to solve a problem, which is useful for students that would otherwise have difficulty solving the problem themselves. For students that like to solve a problem their own way, however, such an approach can be considered too strict. The teaching assistants always agreed with the students in question however, so it is not much of a problem, even though Personal Prof might give unrelated feedback in such a case.

  > "Sometimes we didn't agree with the feedback, but that's because the assignment listed one specific way to solve the problem that you had to use, but that we didn't agree with. So that's not really Personal Prof's fault but more the assignment's fault." - Student 7

- The lectures and tutorial were out of sync

  One of the students mentioned that the tutorial was one week behind schedule compared to the lectures at some point in the course, but that it is no longer the case.

  > "One thing about the course in general was that at some point in the course, the lectures and the tutorial were kind of out of sync. So the lecture was about this topic and the tutorial was about the topic that we had the week before, so they were all kind of not synchronized in a way, so that was kind of weird. But I think it's good now." - Student 6

- The lectures were not recorded

  Due to the university being closed, all lectures are recorded now, but this student would prefer that to always be the case.

  > "Now it doesn't matter, but when the lectures weren't recorded, that was kind of annoying. Especially because the lecture room we were in was too small for all participants, and then they wouldn't even record it." - Student 6

- The slides contain a lot of code and little explanation

  One of the students has had problems with a lack of structure in the slides of the course, as they contain many slides with code and little explanation. As a possible solution, he proposed that the students could use their laptop to try things during the lecture, where the lecturer guides them.

  > "The slides are very overwhelming and very unexplaining. This week was about streams, and this week it was good because it was split up into starting, intermediate and terminal streams, which were explained well. But the other weeks it was totally chaos, you don't know what he's talking about, what he wants to tell me. There are just way too many slides with code." - Student 8

Compliments

- Even when the assignment was well done, the assistants still give points of improvement

  One thing that multiple students seemed to appreciate was that teaching assistants always tried to give points of improvement, even if the assignment was well done.

> "The feedback from the TAs was mostly just positive for my group, but often we would get little suggestions like 'this is fine but you could also do it that way which would be more efficient or cleaner or whatever'. So in general I think the feedback is pretty useful." - Student 6

> "What I did find useful is that if you get a good mark, that you are told 'okay, it was very good, but maybe you can change this and this', so they give you extra options or they explain little things that maybe weren't completely right. I think that's nice." - Student 7

5.4.4 Rubric Grades

	Names	Idiom	Decomposition	Comments	Format	Expression
6	4/4	4/4	4/4	1/4	4/4	4/4
7	4/4	4/4	3/4	1/4	4/4	4/4
8	4/4	4/4	4/4	2/4	4/4	3/4

Table 5.5: Graded work of the third iteration of students, according to the rubric.

5.5 Combining Results

5.5.1 Accuracy

From the grades found in table 5.2 and the more detailed grades described in each iteration, we see that the code quality of these eight students was quite good. Most points were lost due to a lack of comments, but the students were never asked to comment their code, and it is not much of a problem here as the teaching assistants know the assignments very well. On rare occasions, expressions could be simplified and functions could be clarified by splitting them up, but overall there is little to comment on. This corresponds to the positive feedback that Personal Prof gave to the students most of the time. Moreover, on several occasions, the students mentioned quality issues that would have decreased their style grade if Personal Prof had not warned them. This includes issues with naming conventions, issues with access modifiers, and missing classes.

5.5.2 Usage

As can be seen from the 'Personal Prof' sections, each of the eight students said they always read the automatically generated feedback. Six of them

always fixed all issues Personal Prof indicated, one only fixed the larger issues if they lacked time, but still learned from the smaller issues mentioned, and one mentioned fixing the issues only if they agreed that it was better. Therefore, each of them used the automatic feedback to their advantage: they either critically looked at the feedback and decided which was the better way to implement something, or they accepted Personal Prof's feedback as the better solution and learned from the process of improving it, both of which are desirable.

5.5.3 Opinion

In the 'Complaints' sections, and in the overview of table 5.1, we see that the students named a total of seven complaints about Personal Prof. None of these points were mentioned more than once, and two of them were resolved during one of the Personal Prof adaptation phases. Two other points described very specific cases around intended compile time errors and testing. This leaves us with the possibility of students not being able to find the feedback, which can be resolved with a simple announcement, the possibility that Personal Prof might sometimes take much longer to upload feedback, which might have something to do with a server issue, and the fact that Personal Prof usually takes about five minutes to upload feedback, which was a conscious design choice. Overall, the students had quite few complaints about the new software. On the other hand, the 'Compliments' sections show a total of thirteen cases where Personal Prof was praised, often more than once per category. Helping students fix small code issues was by far the most well-received quality of Personal Prof, and the general tone towards the new tool was enthusiastic as well.

Chapter 6

Conclusion

6.1 Research Subquestions

How well does the feedback tool recognize quality flaws?

The tool gave positive feedback in cases where the rubric of Stegeman et al. (2016) was positive as well, and it gave negative feedback in cases where the rubric would have been negative. In that sense, Personal Prof and the rubric agree to a certain extent on what is, and what is not high-quality code. Therefore we conclude that Personal Prof can accurately recognize software quality issues corresponding to the 'names', 'idiom' and 'decomposition' rows of the rubric.

What do the students do with the generated feedback?

The students always read the feedback they receive from the tool, and the majority of them fix all quality issues it mentions. Some choose to ignore part of the feedback in case they consider their own solution high-quality already. We can conclude that the students use the feedback to improve their own code, and as a second opinion on its quality.

How do the students value the automated feedback?

The students mentioned relatively few complaints about the tool, and some of them have already been resolved. Moreover, the tool received a large number of compliments as well, mostly on how it helps the students fix small issues with their code. Overall, we can conclude that the students' response to the automated feedback is positive.

6.2 Research Question

How useful is automatic, style-related feedback to students of object-oriented programming?

Because the tool appears to be accurate, and the students make extensive use of Personal Prof and respond positively to the feedback it gives, we can conclude that automated feedback on code quality can be quite useful to object-oriented programming students.

Chapter 7

Discussion

7.1 Reflection on Findings

7.1.1 Quality Perception

While the number of quality aspects from the rubric named during the interviews remained more or less constant throughout the research, we did notice a slight increase in confidence when the students talked about the quality of their code in later iterations. Especially the snake assignment seemed to trigger the students to think about issues like decomposition and modularization more. Moreover, as can be seen from the participants' feedback: issues with access modifiers, naming conventions, tests and redundant classes mostly occurred during the first few weeks of the course, and quickly became less common. Since Personal Prof was the only source of feedback on these issues, as they were fixed by the participants before they were graded by an assistant, it could be that the students learned this from Personal Prof. However, as this is a beginners course, it can also be explained by the students improving their programming abilities.

7.1.2 Meeting Expectations

We noticed that there was only one case where a student expected Personal Prof to filter an issue that the teaching assistant lowered their grade for, namely using public modifiers for interface methods. This should indeed have been noticed by Personal Prof, and was solved later. While the lack of this kind of feedback does not guarantee that Personal Prof indeed filters all the issues it should filter, it does likely mean that these students did not mind the cases where it might have failed to do so. Moreover, every student who had trouble with the feedback tool had the opportunity to talk about in an interview. The lack of participants might indicate that either no one was particularly interested, or that there were few problems with it in general. Either way, Personal Prof seems to be meeting the students' expectations,

whether those were high or low.

7.1.3 Issue Recognition

We concluded that Personal Prof can accurately indicate issues related to names, idiom or decomposition, but the rules related to names and decomposition are hard-coded per assignment. This works well in cases where the student follows the assignment description closely, which most students do. However, different names or different classes can be just as good as the intended solution quality-wise, but these are harder for Personal Prof to pick up on. While this does not pose a problem for the majority of students, it is important to keep this in mind in cases where assignments can be solved in many ways.

7.1.4 Course Feedback

The students named a large number of points of improvement for the course in general, as can be seen from the 12 'Points of improvement' listed in chapter 5. About half of these points can be addressed with little time investment by:

- mentioning that code should be split in separate files in Java,
- listing which classes Personal Prof expects in the assignment description,
- clarifying that positive feedback from Personal Prof does not guarantee that the teaching assistant is positive as well,
- encouraging students to solve assignments in different ways than described, and
- making sure the lectures and tutorials stay in sync.

The other half, however, is more difficult to resolve. Making teaching assistants agree on what high-quality code entails is difficult, as some aspects of software quality can be subjective. Other points are solvable but require a large time investment. What stood out to us was that the code 'The feedback of the assistant is too brief' gradually became more common throughout the iterations. All of the students mentioned this in the final iteration. This may be caused by the fact that the students of that iteration received little feedback in general due to them doing the assignments well, as can be seen from the code 'Even when the assignment was well done, the assistants still give points of improvement', which applied to two of them. Another explanation could be that the assignments gradually get more difficult and require more feedback, while the time investment for the grading assistants cannot grow as much.

7.2 Reflection on Methods

7.2.1 Number of Participants

The first thing to discuss about the methodology is the number of participants. Due to this being a qualitative analysis, the number of participants is quite small. The conclusions we pulled from the collected data applies to the eight students we interviewed, and while there are undoubtedly other people following the object oriented programming course with similar experiences, we cannot generalize this small sample size to the entire group of over 400 students.

7.2.2 Participant Bias

The convenience sampling method used for this research naturally attracts students that either have a strong opinion about Personal Prof, or were very interested in the snack as reward. This may have affected the results of the research to be more extreme in the former case, or more indifferent in the latter. We do not consider this a problem for this research, as we are confident the students did not do it solely for the reward, and more extreme opinions still allow us to determine how Personal Prof affects the students and how it can be improved. However, it should be noted regardless. If the university was not closed due to the coronavirus, we would have preferred our original plan of randomly selecting students during the tutorial sessions, as this would result in fewer issues with the number of participants, less time in between interviews of the same iteration, and less bias in general.

7.2.3 Zero Measurement

Due to Personal Prof being active for the entire duration of the course, we were unable to perform a proper zero measurement. Therefore, we cannot guarantee that the measured positive effects were definitively caused by Personal Prof, rather than another source. For example, it is possible that the feedback of the teaching assistants improved now that style-related issues of students' code can partially be filtered before grading, which could explain a better understanding of code quality, although the participants often considered the feedback of the assistants too brief. Another explanation would be that the students simply became more adept at programming, causing them to understand how certain quality issues might arise. Moreover, we do not know if the students now have a better understanding of the style-related issues that Personal Prof commented on than they normally would.

7.3 Implications for Practice

Despite the aforementioned uncertainties, Personal Prof proved to be a promising enhancement to the feedback loop of object oriented programming regardless, and will continue to be used and improved the coming years. Now that its practical use has been tested, it can be deployed on other schools or universities, provided they implement rulesets for their coding assignments. This could significantly improve the tool as well, as there would be more data to learn from. Moreover, with a few adjustments, Personal Prof can also be used to judge assignments written in languages other than Java using the same rulesets, making it easily accessible for other courses on the Radboud University or other institutions. The professor of the imperative programming course has expressed interest in using the software, which will likely be the next step.

7.4 Future Research

One of the shortcomings of this research is the inability to generalize the outcome to a larger group of people. The small number of participants and the possibility of bias due to the sampling method makes it difficult to reason about the entire population of students. However, the qualitative analysis did provide us with an extensive list of codes describing possible difficulties, behaviors and opinions. Now that we know more about the possible effects static analysis tools can have on students, we can use these results to conduct quantitative research to give us better insight in what codes occur more often. This can be done through a questionnaire, for example. Such data can be used to see if certain problems occur often, if previous problems have been fixed successfully, or if the quality perception of students improves over time, which was not directly visible from this research.

Furthermore, it would be interesting to see whether or not static analysis tools can give useful feedback on all quality aspects from the rubric of Stegeman et al. (2016). This has proved to be a challenge before, but previous attempts had little information on what the programmer was implementing. An approach where the tool has access to the details of the assignment might make this goal more accessible. Such a tool would not be as useful for everyday programming tasks, which is why this field of static analysis receives little attention, but for educational purposes it can be helpful.

Bibliography

Akour, M., & Falah, B. (2016). Application domain and programming language readability yardsticks. In *2016 7th international conference on computer science and information technology (csit)* (p. 1-6).

ATLAS.ti Scientific Software Development GmbH. (1993). *Atlas.ti.* https://atlasti.com/. (Accessed: 2020-05-19)

Cardell-Oliver, R., Zhang, L., Barady, R., Lim, Y. H., Naveed, A., & Woodings, T. (2010, 4). Automated feedback for quality assurance in software engineering education. In *2010 21st australian software engineering conference* (p. 157-164). doi: 10.1109/ASWEC.2010.24

CWI Amsterdam. (2014). *Rascal.* https://www.rascal-mpl.org/. (Accessed: 2020-03-26)

Eclipse Foundation. (2001). *Eclipse.* https://www.eclipse.org/. (Accessed: 2020-06-15)

Gosling, J., Joy, B., Steeleand G. Bracha, G., & Buckley, A. (2015). In *The java language speci cation, java se 8 edition* (pp. 272, 288-289). Oracle.

Keuning, H., Heeren, B., & Jeuring, J. (2017). Code quality issues in student programs. In *Proceedings of the 2017 acm conference on innovation and technology in computer science education* (p. 110–115). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3059009.3059061 doi: 10.1145/3059009.3059061

King's College London. (2009). *Bluej.* https://www.bluej.org/. (Accessed: 2020-06-03)

Oates, B. J. (2006). In *Researching information systems and computing.* SAGE Publications.

Sadou, N., Tamzalit, D., & Oussalah, M. (2005). A unified approach for software architecture evolution at different abstraction levels. In *Eighth international workshop on principles of software evolution (iwpse'05)* (p. 65-68).

Sasaki, Y., Higo, Y., & Kusumoto, S. (2013). Reordering program statements for improving readability. In *2013 17th european conference on software maintenance and reengineering* (p. 361-364).

Stegeman, M., Barendsen, E., & Smetsers, S. (2016). Designing a rubric for feedback on code quality in programming courses. In *Proceedings of the 16th koli calling international conference on computing education research* (p. 160–164). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2999541.2999555 doi: 10.1145/2999541.2999555

Wiese, E. S., Yen, M., Chen, A., Santos, L. A., & Fox, A. (2017). Teaching students to recognize and implement good coding style. In *Proceedings of the fourth (2017) acm conference on learning @ scale* (p. 41–50). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3051457.3051469 doi: 10.1145/3051457.3051469

Appendix A

Interview

Students' views on code quality

1. What are, in your opinion, the three most important aspects of code quality?
2. Why do you consider these aspects important?

Personal Prof's performance

3. Did you use the Personal Prof automatic code quality testing tool?
   (a) What feedback did you receive and what did you do with it?
   (b) Did you get feedback that you don't agree with or feedback that was unclear?

Feedback effectiveness

4. Regarding all feedback you got for this course, what did you consider particularly useful and which points of improvement would you suggest?

Appendix B

Consent Form

Name research: The Effect of Personal Prof on the Feedback of Object Orientation

Researcher: Rick van der Wal

I, a bachelor student of the Radboud University, am conducting research on the effectiveness of the Personal Prof automatic code quality feedback tool, under supervision of Sjaak Smetsers. This interview is held with several students after getting feedback on the 6^{th}, 8^{th} or 10^{th} assignment. This form provides an explanation on this research.

Purpose and execution of the research

The purpose of this research is to determine the effect of Personal Prof on the quality of feedback given to first year students for object orientation. As of this year, Personal Prof is used to provide style-related feedback on assignments before they are graded by student assistants, allowing the students to improve their code before they get a final grade. The participants of this research will receive several questions to determine how well Personal Prof works in this context. Their answers will be recorded. Additionally, their assignment will be graded on style. This data will be used to improve Personal Prof for future iterations.

Risks and inconveniences

This research poses no threat regarding your health or safety.

Confidentiality of the research data

This research acquires data in the form of audio recordings and style-grades of assignments. The spoken text from the audio recordings will be written

down and anonymized. Along with the grades, this anonymous data will be included in the resulting thesis of this research, which will be accessible by everyone. All data will be stored securely on an encrypted ssd and will be destroyed when the research is done at the end of this semester.

Voluntary service

Participation in this research is completely voluntary. You can choose to withdraw at any moment. All data gathered so far will then be destroyed.

Questions about the research

Should you have any questions, feel free to ask at any time. You can also contact me by email: R.vanderWal@student.ru.nl

Thank you for participating in this research!

Sincerely,

Rick van der Wal

Participant declaration

I received an explanation about the purpose of this research. I was allowed to ask questions. My participation in this research is on a voluntary basis. I understand that I may choose to withdraw from this research at any moment. I understand how the data of this research is stored and how it will be used. I agree to participate in this research.

Name:

Date of birth:

Date:

Signature:

Researcher declaration

I declare that I correctly informed the participant and will adhere to the guidelines for researchers.

Name:

Date:

Signature:

Appendix C

Rubric

LEVEL	1	2	3	4
names	names appear unreadable, meaningless or misleading	names accurately describe the intent of the code, but can be incomplete, lengthy, misspelled or inconsistent use of casing	names accurately describe the intent of the code, and are complete, distinctive, concise, correctly spelled and consistent use of casing	all names in the program use a consistent vocabulary
headers	headers are generally missing or descriptions are redundant or obsolete; use mixed languages or are misspelled	header comments are generally present; summarize the goal of parts of the program and how to use those; but may be somewhat inaccurate or incomplete	header comments are generally present; accurately summarize the role of parts of the program and how to use those; but may still be wordy	header comments are generally present; contain only essential explanations, information and references
comments	comments are generally missing, redundant or obsolete; use mixed languages or are misspelled	comments explain code and potential problems, but may be wordy	comments explain code and potential problems, are concise	comments are only present where strictly needed
layout	old commented out code is present or lines are generally too long to read	positioning of elements within source files is not optimized for readability	positioning of elements within source files is optimized for readability	positioning of elements is consistent between files and in line with platform conventions
formatting	formatting is missing or misleading	indentation, line breaks, spacing and brackets highlight the intended structure but erratically	indentation, line breaks, spacing and brackets consistently highlight the intended structure	formatting makes similar parts of code clearly identifiable
flow	there is deep nesting; code performs more than one task per line; unreachable code is present	flow is complex or contains many exceptions or jumps; parts of code are duplicate	flow is simple and contains few exceptions or jumps; duplication is very limited	in the case of exceptions or jumps, the most common path through the code is clearly visible
idiom	control structures are customized in a misleading way	choice of control structures is inapproriate	choice of control structures is appropriate; reuse of library functionality may be limited	reuse of library functionality and generic data structures where possible
expressions	expressions are repeated or contain unnamed constants	expressions are complex or long; data types are inappropriate	expressions are simple; data types are appropriate	expressions are all essential for control flow
decomposition	most code is in one or a few big routines; variables are reused for different purposes	most routines are limited in length but mix tasks; routines share many variables instead of having parameters	routines perform a limited set of tasks divided into parts; use of shared variables is limited	routines perform a very limited set of tasks and the number of parameters and shared variables is limited
modularization	most code is in one or a few large modules; or modules are artificially separated	modules have mixed responsibilities, contain many variables or contain many routines	modules have clearly defined responsibilities, contain few variables and a somewhat limited amount of routines	modules are defined such that communication between them is limited

- for each criterion, circle the level that is most representative of the features that are present
- no need to circle a level that is not relevant to the assignment
- level 2 implies that the features in level 1 are not present, level 4 implies that the features in level 3 are also present

Appendix D

Snake assignment

Assignment Snake

Object Orientation

Spring 2020

1 Snake

In this assignment you implement a JavaFX snake game. The snake can be turned left or right using the keyboard while it also moves forward at a quick pace. The user has to direct the snake towards the food. When the snake eats the food, its length increases by one and the food item is moved to a random location. Clicking the screen moves the food to the mouse position. The game ends when the snake collides with its own body or the playing field boundaries.

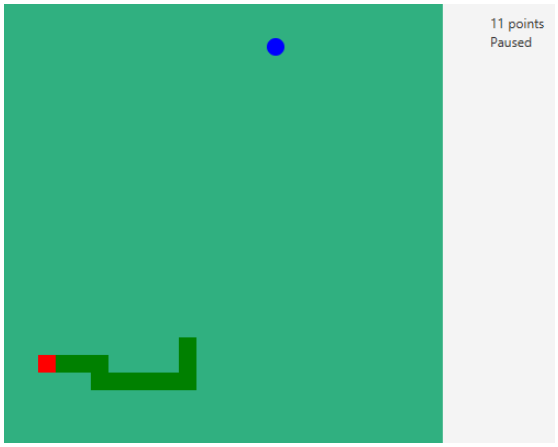

Figure 1: Example of a finished product. The snake is made up of green rectangles and a red head. The food item is a blue circle.

2 Learning Goals

After doing this exercise you should be able to:

- Use a `Timeline` to periodically execute code.
- Use keyboard handlers to process input.
- Know how and why to separate the game data, user interface, and input handler in GUI applications

3 Problem Sketch

The core functionality of this exercise consists of three major parts: the game data, the user interface and the input handler. The game data consists of all game logic, while the user interface handles all the visual representations on the screen and the input handler handles key presses and mouse clicks. For this assignment it is important that these three concepts are separated well. This implies that the game logic, the JavaFX objects and the EventHandlers are in separate modules/classes, using property binding to communicate between them. The project template on Brightspace should help achieving this.

While it is quite common for games to make game objects directly drawable, this practice is considered unidiomatic for user interface design, which is why, for the purposes of this assignment, you should try to keep them separated. There are other reasons for this, but one very practical reason is that many programming languages can run on multiple different platforms, but the libraries that handle visuals or input are often platform-specific (for example, the JavaFX library handles both visuals and input for us, but it doesn't work on Android, even though Android runs Java). When you port software to a different platform, or want to switch certain libraries for another reason, you only have to change the modules that use those libraries, so keeping them well-contained reduces the amount of work and increases the flexibility of your software.

4 Classes

The project template on Brightspace contains a total of 8 classes:

- `Main` creates a new JavaFX application
- `World` represents the complete state of one snake game.
- `Segment` represents the location of one segment of the snake.
- `Snake` represents the head of the snake and keeps track of all body segments.
- `Food` represents the location of the food item.
- `InputHandler` contains all functionality regarding keyboard- and mouse input and adapts the given world instance accordingly.
- `SnakeGame` is a JavaFX Pane that handles displaying the given world instance
- `Direction` is a simple enumeration you should use to indicate the direction in which the snake is currently moving. It contains two helper functions to rotate any direction 90 degrees to the left or right.

5 Assignment

For this assignment, you should implement the following functionality:

5.1 Timeline

The movement of the snake should be triggered by a `Timeline` every `DELAY` milliseconds with an infinite cycle count. The timeline should play or pause depending on the game's `running` state.

5.2 Movement

In every step, you should calculate the potential new position of the head of the snake. There are three different possibilities for this:

- If the snake head would collide with its own body or the field boundaries, it's game over and the animation should stop. You don't have to implement restarting the game.
- If the snake head would collide with the food, the snake should grow in length and the food should be moved to a random location.
- Otherwise, the snake moves to the new position, dragging its body with it.

5.3 Graphics

There is no explicit function for drawing the game elements, all visible elements should be JavaFX objects, such as `Circle` or `Rectangle`, located in `SnakeGame`. When the game data changes, the user interface should be updated automatically by binding the locations of the JavaFX objects to the location properties of the objects. Keep in mind that the coordinates of the visual representations change by a larger amount than the game coordinates of objects, due to them being scaled.

Keeping positions of shapes in sync with the game data can be accomplished using property bindings. But if the snake eats the food, it grows an additional segment, for which an additional square must be drawn on the screen. Growing the snake is implemented in the game logic. How should the graphical representation know that it has to create a new square for the new segment? We do not want to implement this functionality in `Snake`, because only `SnakeGame` may contian JavaFX code.

The solution is to implement a callback mechanism in `Snake`. Whenever the snake grows, it calls a registered function with the new segment as argument. `SnakeGame` has to register a function there on program startup, and this function can have JavaFX code. This function must create a new square and bind its location to the segment position.

In particular, this means:

1. In `SnakeGame`, implement `Snake.SnakeSegmentListener`. You can do this in a number of ways, for example by letting SnakeGame itself implement SnakeSegmentListener, or by creating a private inner class that implements SnakeSegmentListener, or by using a lambda expression.
2. In the program initialization code, register it to `Snake`'s list of listeners.
3. In `Snake`, call `onNewSegment(segment)` every time a new segment is created.

This pattern, called the *observer pattern*, is a simplified version of how property binding works behind the scenes. It allows us to accurately represent every object in `SnakeGame` without breaking the barrier between user interface and game data. In this exercise there will only be one listener, but in general there could be many. That's why `Snake` has a list of listeners, even though only `SnakeGame` will be listening.

5.4 Controls

Keyboard input should be handled in `InputHandler`. When the user presses `'s'`, the program should pause if it wasn't paused already and resume otherwise. You can use the `pause()` and `play()` functions from `Timeline` for this. When the user presses `'a'` or `'d'`, the snake should turn left or right, respectively.

5.5 Mouse

Mouse input should be also handled in `InputHandler`. When the user clicks the screen, the food should be teleported to its location, provided that it is on the field.

5.6 User Interface

There should be two lines of text on the screen. One should display "x points", where x should be the current score. The other should indicate whether or not the game is paused. If the game hasn't started yet, it should read "Press 's' to start" instead. The user interface should update automatically when the game state changes using property binding.

6 Finished Product

After implementing the features above, verify the following:

1. The snake moves forward every `DELAY` milliseconds.
2. Pressing `'a'` or `'d'` makes the snake turn left or right, respectively, while pressing `'s'` toggles between pause and running.
3. Clicking the screen teleports the food to the mouse location.
4. Eating the food increases the length of the snake by one and teleports the food to a random new location.
5. Biting your own tail or the wall pauses the game.
6. The user interface displays up-to-date information about the score and whether or not the game is paused.

If you get an error message about the missing JavaFX library, create a new global library of that name, and add all JavaFX jar files to it. Detailed instructions on how to do that can be found in the pie charts assignment text.

7 Submit Your Project

To submit your project, follow these steps.

1. Use the **project export** feature of NetBeans to create a zip file of your entire project: File → Export Project → To ZIP.
2. **Submit this zip file on Brightspace**. Do not submit individual Java files. Do not submit any other archive format. Only one person in your group has to submit it. Submit your project before the deadline, which can be found on Brightspace.

Appendix E

Snake Template

snake–template/Main.java

```
1  package snake;
2
3  import javafx.application.Application;
4  import javafx.geometry.Insets;
5  import javafx.scene.Scene;
6  import javafx.scene.layout.BorderPane;
7  import javafx.scene.layout.Pane;
8  import javafx.stage.Stage;
9
10 public class Main extends Application {
11
12     @Override
13     public void start(Stage primaryStage) {
14         World world = new World(25);
15
16         BorderPane root = new BorderPane();
17         SnakeGame game = new SnakeGame(world);
18         Pane ui = SnakeGame.createUserInterface(world);
19
20         game.setStyle("-fx-background-color: #30B080;");
21         ui.setPadding(new Insets(10));
22
23         root.setLeft(game);
24         root.setRight(ui);
25
26         Scene scene = new Scene(root);
27
28         InputHandler inputHandler = new InputHandler(world);
29
30         scene.setOnKeyPressed(inputHandler.getKeyHandler());
31         scene.setOnMouseClicked(inputHandler.getMouseHandler());
32
33         primaryStage.setTitle("Snake");
34         primaryStage.setScene(scene);
35         primaryStage.show();
36     }
```

```
37
38      public static void main(String[] args) {
39          launch(args);
40      }
41  }
```

snake–template/InputHandler.java

```
1   package snake;
2
3   import javafx.event.EventHandler;
4   import javafx.scene.input.KeyEvent;
5   import javafx.scene.input.MouseEvent;
6
7   /**
8    * Handles controls of a snake game, where the 'a' and 'd' keys
         can be used to move and 's' (un)pauses the game
9    */
10  public class InputHandler {
11
12      private final EventHandler<KeyEvent> keyHandler;
13      private final EventHandler<MouseEvent> mouseHandler;
14
15      public InputHandler(World world) {
16          Snake snake = world.getSnake();
17
18          keyHandler = keyEvent -> {
19              // TODO: Implement controls
20              keyEvent.consume();
21          };
22
23          mouseHandler = mouseEvent -> {
24              // TODO: Implement mouse
25              mouseEvent.consume();
26          };
27      }
28
29      public EventHandler<KeyEvent> getKeyHandler() {
30          return keyHandler;
31      }
32
33      public EventHandler<MouseEvent> getMouseHandler() {
34          return mouseHandler;
35      }
36  }
```

snake–template/SnakeGame.java

```
1   package snake;
2
3   import javafx.scene.control.Label;
4   import javafx.scene.layout.Pane;
5   import javafx.scene.layout.VBox;
6
```

```
7  /**
8   * A JavaFX Pane that displays the snake game represented by the
        given world
9   */
10 public class SnakeGame extends Pane {
11
12     public static final int SCALE = 16;
13
14     public SnakeGame(World world) {
15         setPrefSize(world.getSize() * SCALE, world.getSize() *
               SCALE);
16
17         // TODO: Implement graphics
18     }
19
20     public static Pane createUserInterface(World world) {
21         VBox ui = new VBox();
22
23         Label scoreText = new Label();
24         Label runningText = new Label("Press 's' to start");
25
26         // TODO: Implement user interface
27
28         ui.getChildren().addAll(scoreText, runningText);
29
30         return ui;
31     }
32 }
```

snake–template/World.java

```
1  package snake;
2
3  import javafx.beans.property.BooleanProperty;
4  import javafx.beans.property.IntegerProperty;
5  import javafx.beans.property.SimpleBooleanProperty;
6  import javafx.beans.property.SimpleIntegerProperty;
7
8  import java.util.Random;
9
10 /**
11  * World keeps track of the state of a snake game
12  */
13 public class World {
14
15     public final static int DELAY = 200;
16
17     private final int size;
18
19     private final Snake snake;
20     private final Food food;
21
22     private final Random random = new Random();
23
```

```
24      private final BooleanProperty running = new
            SimpleBooleanProperty(false);
25
26      private final IntegerProperty score = new
            SimpleIntegerProperty(0);
27
28      public World(int size) {
29          this.size = size;
30
31          snake = new Snake(size / 2, size / 2, this);
32          food = new Food();
33
34          // TODO: Implement timeline
35
36          moveFoodRandomly();
37      }
38
39      public void moveFoodRandomly() {
40          do {
41              food.moveTo(random.nextInt(size), random.nextInt(
                    size));
42          } while (snake.isAt(food.getX(), food.getY()));
43      }
44
45      public void endGame() {
46          running.set(false);
47      }
48
49      public void setRunning(boolean running) {
50          this.running.set(running);
51      }
52
53      public void setScore(int score) {
54          this.score.set(score);
55      }
56
57      public boolean isRunning() {
58          return running.get();
59      }
60
61      public int getSize() {
62          return size;
63      }
64
65      public int getScore() {
66          return score.get();
67      }
68
69      public Snake getSnake() {
70          return snake;
71      }
72
73      public Food getFood() {
74          return food;
```

```
75      }
76
77      public BooleanProperty getRunningProperty() {
78          return running;
79      }
80
81      public IntegerProperty getScoreProperty() {
82          return score;
83      }
84  }
```

snake–template/Segment.java

```
1   package snake;
2
3   import javafx.beans.property.IntegerProperty;
4   import javafx.beans.property.SimpleIntegerProperty;
5
6   /**
7    * Represents one body part of a snake
8    */
9   public class Segment {
10
11      private final IntegerProperty x, y;
12
13      public Segment(int x, int y) {
14          this.x = new SimpleIntegerProperty(x);
15          this.y = new SimpleIntegerProperty(y);
16      }
17
18      public void setPosition(int x, int y) {
19          this.x.setValue(x);
20          this.y.setValue(y);
21      }
22
23      public int getX() {
24          return x.get();
25      }
26
27      public int getY() {
28          return y.get();
29      }
30
31      public IntegerProperty getXProperty() {
32          return x;
33      }
34
35      public IntegerProperty getYProperty() {
36          return y;
37      }
38  }
```

snake–template/Snake.java

```java
package snake;

import java.util.LinkedList;
import java.util.List;

/**
 * Snake consists of segments, where this head segment keeps track of the other body segments
 */
public class Snake extends Segment {

    public interface SnakeSegmentListener {
        public void onNewSegment(Segment segment);
    }

    private Direction direction = Direction.RIGHT;

    private final World world;

    private final List<Segment> body = new LinkedList<>();

    private final List<SnakeSegmentListener> listeners = new LinkedList<>();

    public Snake(int x, int y, World world) {
        super(x, y);
        this.world = world;
    }

    public void move() {
        int newX = getX() + direction.getDX();
        int newY = getY() + direction.getDY();

        // TODO: Implement movement
    }

    public void addListener(SnakeSegmentListener listener) {
        listeners.add(listener);
    }

    public void setDirection(Direction newDirection) {
        direction = newDirection;
    }

    public boolean isAt(int x, int y) {
        for (Segment segment : body) {
            if (segment.getX() == x && segment.getY() == y) {
                return true;
            }
        }

        return false;
    }

```

```
53      public Direction getDirection() {
54          return direction;
55      }
56  }
```

snake–template/Food.java

```
1   package snake;
2
3   import javafx.beans.property.IntegerProperty;
4   import javafx.beans.property.SimpleIntegerProperty;
5
6   public class Food {
7
8       private final IntegerProperty x = new SimpleIntegerProperty
            (), y = new SimpleIntegerProperty();
9
10      public void moveTo(int x, int y) {
11          this.x.set(x);
12          this.y.set(y);
13      }
14
15      public int getX() {
16          return x.get();
17      }
18
19      public int getY() {
20          return y.get();
21      }
22
23      public IntegerProperty getXProperty() {
24          return x;
25      }
26
27      public IntegerProperty getYProperty() {
28          return y;
29      }
30  }
```

snake–template/Direction.java

```
1   package snake;
2
3   public enum Direction {
4       UP(0, -1),
5       RIGHT(1, 0),
6       DOWN(0, 1),
7       LEFT(-1, 0);
8
9       private final int dX, dY;
10
11      private Direction(int dX, int dY) {
12          this.dX = dX;
13          this.dY = dY;
```

```
14      }
15
16      public int getDX() {
17          return dX;
18      }
19
20      public int getDY() {
21          return dY;
22      }
23
24      public Direction rotateLeft() {
25          switch(this) {
26              case UP: return LEFT;
27              case LEFT: return DOWN;
28              case DOWN: return RIGHT;
29              default: return UP;
30          }
31      }
32
33      public Direction rotateRight() {
34          switch(this) {
35              case UP: return RIGHT;
36              case RIGHT: return DOWN;
37              case DOWN: return LEFT;
38              default: return UP;
39          }
40      }
41  }
```

Appendix F

Snake Example Solution

snake–solution/Main.java

```
1  package snake;
2
3  import javafx.application.Application;
4  import javafx.geometry.Insets;
5  import javafx.scene.Scene;
6  import javafx.scene.layout.BorderPane;
7  import javafx.scene.layout.Pane;
8  import javafx.stage.Stage;
9
10 public class Main extends Application {
11
12     @Override
13     public void start(Stage primaryStage) {
14         World world = new World(25);
15
16         BorderPane root = new BorderPane();
17         SnakeGame game = new SnakeGame(world);
18         Pane ui = SnakeGame.createUserInterface(world);
19
20         game.setStyle("-fx-background-color: #30B080;");
21         ui.setPadding(new Insets(10));
22
23         root.setLeft(game);
24         root.setRight(ui);
25
26         Scene scene = new Scene(root);
27
28         InputHandler inputHandler = new InputHandler(world);
29
30         scene.setOnKeyPressed(inputHandler.getKeyHandler());
31         scene.setOnMouseClicked(inputHandler.getMouseHandler());
32
33         primaryStage.setTitle("Snake");
34         primaryStage.setScene(scene);
35         primaryStage.show();
36     }
```

```
37
38      public static void main(String[] args) {
39          launch(args);
40      }
41  }
```

snake–solution/InputHandler.java

```
1   package snake;
2
3   import javafx.event.EventHandler;
4   import javafx.scene.input.KeyEvent;
5   import javafx.scene.input.MouseEvent;
6
7   /**
8    * Handles controls of a snake game, where the 'a' and 'd' keys
         can be used to move and 's' (un)pauses the game
9    */
10  public class InputHandler {
11
12      private final EventHandler<KeyEvent> keyHandler;
13      private final EventHandler<MouseEvent> mouseHandler;
14
15      public InputHandler(World world) {
16          Snake snake = world.getSnake();
17
18          keyHandler = keyEvent -> {
19              switch (keyEvent.getCode()) {
20                  case A:
21                      snake.setDirection(snake.getDirection().
                            rotateLeft());
22                      break;
23                  case D:
24                      snake.setDirection(snake.getDirection().
                            rotateRight());
25                      break;
26                  case S:
27                      world.setRunning(!world.isRunning());
28                      break;
29              }
30              keyEvent.consume();
31          };
32
33          mouseHandler = mouseEvent -> {
34              int x = (int) (mouseEvent.getX() / SnakeGame.SCALE);
35              int y = (int) (mouseEvent.getY() / SnakeGame.SCALE);
36
37              if(x >= 0 && y >= 0 && x < world.getSize() && y <
                    world.getSize()) {
38                  world.getFood().moveTo(x, y);
39              }
40
41              mouseEvent.consume();
42          };
```

```
43      }
44
45      public EventHandler<KeyEvent> getKeyHandler() {
46          return keyHandler;
47      }
48
49      public EventHandler<MouseEvent> getMouseHandler() {
50          return mouseHandler;
51      }
52  }
```

snake–solution/SnakeGame.java

```
1   package snake;
2
3   import javafx.geometry.Insets;
4   import javafx.scene.control.Label;
5   import javafx.scene.layout.Pane;
6   import javafx.scene.layout.VBox;
7   import javafx.scene.paint.Color;
8   import javafx.scene.shape.Circle;
9   import javafx.scene.shape.Rectangle;
10
11  /**
12   * A JavaFX Pane that displays the snake game represented by the
         given world
13   */
14  public class SnakeGame extends Pane {
15
16      public static final int SCALE = 16;
17
18      public SnakeGame(World world) {
19          setPrefSize(world.getSize() * SCALE, world.getSize() *
                SCALE);
20
21          // Snake
22
23          Snake snake = world.getSnake();
24
25          Rectangle head = new Rectangle(SCALE, SCALE, Color.RED);
26
27          head.translateXProperty().bind(snake.getXProperty().
                multiply(SCALE));
28          head.translateYProperty().bind(snake.getYProperty().
                multiply(SCALE));
29
30          getChildren().add(head);
31
32          snake.addListener(segment -> {
33              Rectangle body = new Rectangle(SCALE, SCALE, Color.
                    GREEN);
34
35              body.translateXProperty().bind(segment.getXProperty
                    ().multiply(SCALE));
```

```
36              body.translateYProperty().bind(segment.getYProperty
                    ().multiply(SCALE));
37
38              getChildren().add(body);
39          });
40
41          // Food
42
43          Food food = world.getFood();
44
45          Circle circle = new Circle(SCALE / 2F, Color.BLUE);
46
47          circle.translateXProperty().bind(food.getXProperty().
                multiply(SCALE).add(SCALE / 2F));
48          circle.translateYProperty().bind(food.getYProperty().
                multiply(SCALE).add(SCALE / 2F));
49
50          getChildren().add(circle);
51      }
52
53      public static Pane createUserInterface(World world) {
54          VBox ui = new VBox();
55
56          Label scoreText = new Label();
57          Label runningText = new Label("Press 's' to start");
58
59          scoreText.textProperty().bind(world.getScoreProperty().
                asString("%d points"));
60          world.getRunningProperty().addListener((observableValue,
                 aBoolean, t1) -> {
61              if (t1) {
62                  runningText.textProperty().set("");
63              } else {
64                  runningText.textProperty().set("Paused");
65              }
66          });
67
68          ui.getChildren().addAll(scoreText, runningText);
69
70          return ui;
71      }
72  }
```

snake–solution/World.java

```
1   package snake;
2
3   import javafx.animation.Animation;
4   import javafx.animation.KeyFrame;
5   import javafx.animation.Timeline;
6   import javafx.beans.property.BooleanProperty;
7   import javafx.beans.property.IntegerProperty;
8   import javafx.beans.property.SimpleBooleanProperty;
9   import javafx.beans.property.SimpleIntegerProperty;
```

```
10 import javafx.util.Duration;
11
12 import java.util.Random;
13
14 /**
15  * World keeps track of the state of a snake game
16  */
17 public class World {
18
19     public final static int DELAY = 200;
20
21     private final int size;
22
23     private final Snake snake;
24     private final Food food;
25
26     private final Random random = new Random();
27
28     private final BooleanProperty running = new
           SimpleBooleanProperty(false);
29
30     private final IntegerProperty score = new
           SimpleIntegerProperty(0);
31
32     public World(int size) {
33         this.size = size;
34
35         snake = new Snake(size / 2, size / 2, this);
36         food = new Food();
37
38         Timeline timeline = new Timeline(new KeyFrame(Duration.
               millis(DELAY), e -> snake.move()));
39
40         running.addListener((observableValue, aBoolean, t1) -> {
41             if (t1) {
42                 timeline.play();
43             } else {
44                 timeline.pause();
45             }
46         });
47
48         timeline.setCycleCount(Animation.INDEFINITE);
49
50         moveFoodRandomly();
51     }
52
53     public void moveFoodRandomly() {
54         do {
55             food.moveTo(random.nextInt(size), random.nextInt(
                   size));
56         } while (snake.isAt(food.getX(), food.getY()));
57     }
58
59     public void endGame() {
```

```
60          running.set(false);
61      }
62
63      public void setRunning(boolean running) {
64          this.running.set(running);
65      }
66
67      public void setScore(int score) {
68          this.score.set(score);
69      }
70
71      public boolean isRunning() {
72          return running.get();
73      }
74
75      public int getSize() {
76          return size;
77      }
78
79      public int getScore() {
80          return score.get();
81      }
82
83      public Snake getSnake() {
84          return snake;
85      }
86
87      public Food getFood() {
88          return food;
89      }
90
91      public BooleanProperty getRunningProperty() {
92          return running;
93      }
94
95      public IntegerProperty getScoreProperty() {
96          return score;
97      }
98  }
```

snake–solution/Segment.java

```
1   package snake;
2
3   import javafx.beans.property.IntegerProperty;
4   import javafx.beans.property.SimpleIntegerProperty;
5
6   /**
7    * Represents one body part of a snake
8    */
9   public class Segment {
10
11      private final IntegerProperty x, y;
12
```

```
13      public Segment(int x, int y) {
14          this.x = new SimpleIntegerProperty(x);
15          this.y = new SimpleIntegerProperty(y);
16      }
17
18      public void setPosition(int x, int y) {
19          this.x.setValue(x);
20          this.y.setValue(y);
21      }
22
23      public int getX() {
24          return x.get();
25      }
26
27      public int getY() {
28          return y.get();
29      }
30
31      public IntegerProperty getXProperty() {
32          return x;
33      }
34
35      public IntegerProperty getYProperty() {
36          return y;
37      }
38  }
```

snake–solution/Snake.java

```
1   package snake;
2
3   import java.util.LinkedList;
4   import java.util.List;
5
6   /**
7    * Snake consists of segments, where this head segment keeps
         track of the other body segments
8    */
9   public class Snake extends Segment {
10
11      public interface SnakeSegmentListener {
12          public void onNewSegment(Segment segment);
13      }
14
15      private Direction direction = Direction.RIGHT;
16
17      private final World world;
18
19      private final List<Segment> body = new LinkedList<>();
20
21      private final List<SnakeSegmentListener> listeners = new
            LinkedList<>();
22
23      public Snake(int x, int y, World world) {
```

```
24          super(x, y);
25          this.world = world;
26      }
27
28      public void move() {
29          int newX = getX() + direction.getDX();
30          int newY = getY() + direction.getDY();
31
32          if (isAt(newX, newY) || newX < 0 || newY < 0 || newX >=
                world.getSize() || newY >= world.getSize()) {
33              // Bitten itself or the border, game over
34              world.endGame();
35          } else {
36              Food food = world.getFood();
37
38              if (food.getX() == newX && food.getY() == newY) {
39                  // Eating fruit, increment score and add new
                        segment before the head
40                  world.setScore(world.getScore() + 1);
41                  world.moveFoodRandomly();
42
43                  Segment segment = new Segment(getX(), getY());
44
45                  for (SnakeSegmentListener listener : listeners)
                        {
46                      listener.onNewSegment(segment);
47                  }
48
49                  body.add(segment);
50              } else {
51                  // Moving normally, recycle tail and move it
                        before head
52                  if (!body.isEmpty()) {
53                      Segment tail = body.remove(0);
54                      body.add(tail);
55
56                      tail.setPosition(getX(), getY());
57                  }
58              }
59
60              // Move head to new location
61              setPosition(newX, newY);
62          }
63      }
64
65      public void addListener(SnakeSegmentListener listener) {
66          listeners.add(listener);
67      }
68
69      public void setDirection(Direction newDirection) {
70          direction = newDirection;
71      }
72
73      public boolean isAt(int x, int y) {
```

```
74          for (Segment segment : body) {
75              if (segment.getX() == x && segment.getY() == y) {
76                  return true;
77              }
78          }
79
80          return false;
81      }
82
83      public Direction getDirection() {
84          return direction;
85      }
86  }
```

snake–solution/Food.java

```
1   package snake;
2
3   import javafx.beans.property.IntegerProperty;
4   import javafx.beans.property.SimpleIntegerProperty;
5
6   public class Food {
7
8       private final IntegerProperty x = new SimpleIntegerProperty
            (), y = new SimpleIntegerProperty();
9
10      public void moveTo(int x, int y) {
11          this.x.set(x);
12          this.y.set(y);
13      }
14
15      public int getX() {
16          return x.get();
17      }
18
19      public int getY() {
20          return y.get();
21      }
22
23      public IntegerProperty getXProperty() {
24          return x;
25      }
26
27      public IntegerProperty getYProperty() {
28          return y;
29      }
30  }
```

snake–solution/Direction.java

```
1   package snake;
2
3   public enum Direction {
4       UP(0, -1),
```

```
5       RIGHT(1, 0),
6       DOWN(0, 1),
7       LEFT(-1, 0);
8
9       private final int dX, dY;
10
11      private Direction(int dX, int dY) {
12          this.dX = dX;
13          this.dY = dY;
14      }
15
16      public int getDX() {
17          return dX;
18      }
19
20      public int getDY() {
21          return dY;
22      }
23
24      public Direction rotateLeft() {
25          switch(this) {
26              case UP: return LEFT;
27              case LEFT: return DOWN;
28              case DOWN: return RIGHT;
29              default: return UP;
30          }
31      }
32
33      public Direction rotateRight() {
34          switch(this) {
35              case UP: return RIGHT;
36              case RIGHT: return DOWN;
37              case DOWN: return LEFT;
38              default: return UP;
39          }
40      }
41  }
```