MASTER'S THESIS

Automated layered feedback: refactoring imperative loops to higher-order functions

Wu, E.Z.W. (Erik)

Award date:
2021

Link to publication

AUTOMATED LAYERED FEEDBACK: REFACTORING IMPERATIVE LOOPS TO HIGHER-ORDER FUNCTIONS

by

Erik Zhi Wei Wu

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University, faculty of Science
Master Software Engineering
to be defended publicly on Friday October 15, 2021 at 11:00 AM.

Student number:		
Course code:	IMA0002	
Thesis committee:	dr. ir. Sylvia Stuurman (chairman),	Open University of the Netherlands
	dr. Bastiaan Heeren (supervisor),	Open University of the Netherlands

Abstract

In the coursework of universities, the focus is put on getting a program functionally working, rather than writing high-quality code. As a result, graduates often lack the programming skills that businesses desire. This problem is hard to solve, as providing personalized feedback on the students' program code does not fit within the time constraints of teachers. Intelligent Tutoring Systems (ITS) are introduced to lessen the workload of teachers by providing the students automated feedback.

The Refactor Tutor is an ITS that solely focuses on improving code quality by providing 'imperative' automated layered feedback. The automated layered feedback is represented to the students as hints. The hints are provided in a tree structure, with lower-level hints showing more details. The Refactor Tutor does not provide feedback on refactoring imperative code to functional. However, this is very beneficial, since this improves the code quality and students get familiar with a functional style.

The goal of our research is to extend the Refactor Tutor to also provide this type of feedback of refactoring imperative code to functional. Our scientific contribution is to research whether an ITS can be used to teach a functional style by refactoring imperative code. Our first step towards this type of tutor is to support refactoring imperative loops to the higher-order functions *map* and *filter* as most multiparadigm programming languages support these functions.

We extended the Refactor Tutor to support lambda expressions and the higher-order functions *map* and *filter*. We then set up two exercises and the step-by-step hints to refactor to the higher-order functions. Finally, two teachers were interviewed to compare their feedback with the step-by-step hints.

Both teachers questioned whether students would understand the feedback as it requires a different thinking mindset. However, the teachers could see the benefits from this approach and would adapt the system into their courses. Given that this is a novel approach, more research is needed to understand the effect of such refactorings on students.

Keywords: Refactoring, Code quality, Higher-order functions, Map, Filter

PREFACE

This thesis is the final work of my Master Software Engineering at the Open University of the Netherlands. It includes the documentation of my research, which has been carried out between February until September 2021 and describes the thoughts behind the choices I have made.

The subject of the assignment is ‘Automated Layered Feedback: Refactoring imperative loops to higher-order functions’. The research presents a new way of teaching multi-paradigm programming, specifically from imperative programming to functional programming, with the use of an Intelligent Tutoring System.

I would like to especially thank both my supervisors, Bastiaan Heeren and Sylvia Stuurman, for the valuable support and help throughout my research. I would like to thank Hieke Keuning as well for all the help with the Refactor Tutor.

CONTENTS

LIST OF FIGURES

1 Introduction

Students (or novice programmers) often learn to program using an imperative programming language. This is also seen in the work of Jansen et al. [2018] and Keuning et al. [2017] as they have studied introductory courses at universities. Students start with a course introducing an imperative programming language in which they learn how to write a program. The focus is put on getting the program functionally working, rather than writing high-quality code.

Producing high-quality (functional suitable, performant, compatible, useable, reliable, secure, maintainable, portable[1]) software is beneficial to businesses as it minimizes risk exposure. Graduates may not even know how to write high-quality code since it has always been somewhat neglected in the course work. Jansen et al. [2018] also noticed that it is often very hard for teachers to provide the personalized feedback necessary for students to know how to improve their code.

Automated Assessment (AA) systems and Intelligent Tutoring Systems (ITS) are introduced to lessen the workload of teachers by providing the students automated feedback. AA systems instantly provide students with automated feedback. ITS also provide automated feedback to students but try to do so in a rather more interactive approach. VanLehn [2011] proved in an experiment that ITS are nearly as effective as human tutoring and Narciss [2004] has shown that these tools can have a positive impact on the students' sense of achievement, motivation, etc.

These tools have already been employed by universities worldwide, also in the programming domain. However, Keuning et al. [2018] noticed that most of these tools focus on getting a program functionally working rather than improving the code quality. Therefore, the problem of low-quality code among students still exists.

To solve the problem of low-quality code among students, Keuning et al. [2021] introduce a dedicated ITS, the Refactor Tutor, that focuses on code quality. Students are presented with a working program and receive the task to improve the code quality of that program. The system provides an interactive session between the student by providing them with automated layered feedback. The automated layered feedback is represented to the students as hints. The hints are provided in a tree structure, with lower-level hints showing more details. The hints are similar to what a teacher would provide to a student.

[1]ISO/IEC 25010. URL: `https://iso25000.com/index.php/en/iso-25000-standards/iso-25010`. Accessed on September 3, 2021

The Refactor Tutor provides feedback on Java programs. Nowadays, a lot of modern imperative programming languages such as Java or C# are a so-called *multiparadigm* programming language. The language supports multiple programming paradigms such as imperative and functional programming paradigms. It may be very hard for students to understand when and why to use which programming paradigm as the languages support both. Also, these multiparadigm languages combine imperative code with functional, getting the best of both worlds (or the worst).

The feedback provided by the Refactor Tutor is only gives 'imperative' feedback. It does not provide feedback on refactoring imperative code to functional. This is, however, very beneficial, since we can learn students to go from an imperative style to a functional style where possible. This bridges the gap between learning functional programming with an imperative programming background. Mirolo and Izu [2019] conclude that functional programming requires a different type of perspective from students and that this is not an easy task. Also, Benton and Radziwill [2016] argue that a functional approach is often more readable, modular, and testable, which in turn also improves the code quality of the program.

With the feedback to refactor imperative code to functional, the students can slowly get familiar with functional programming concepts and learn why there is another programming paradigm besides the imperative programming paradigm.

The goal of our research is to extend the Refactor Tutor to also provide this type of feedback of refactoring imperative code to functional. Our scientific contribution is to research whether an ITS can be used to teach a functional style by refactoring imperative code. Our first step towards this type of tutor is to support refactoring imperative loops to higher-order functions as most multiparadigm programming languages support higher-order functions. The work of Keuning et al. [2019] has also shown that teachers would provide the feedback of rewriting imperative loops to higher-order functions. This will also increase the code quality of the program (in line with the goal of the Refactor Tutor) and help students get familiar with functional programming concepts one at a time. Our related work provides evidence that the refactoring is possible, however, there has not been any related work that does this in a 'layered' feedback format. Oftentimes, it is a static analysis tool that refactors the code without providing additional information. Jansen et al. [2018] found that these tools lack the support that some students need to improve their code.

The thesis is structured as follows. We discuss the related work in Chapter 2. In Chapter 3 we highlight the goal of our research and discuss our research questions. We then present in Chapter 4 the current situation of teaching a functional style to students with an imperative programming background, in Chapter 5 the steps taken to implement our refactoring into the Refactor Tutor, in Chapter 6 the step-by-step feedback and in Chapter 7 the step-by-step feedback compared to the teachers' feedback. Lastly, we summarize our conclusions to answer the main research question in Chapter 8 and discuss the future work and important decisions that we made during the process in Chapter 9.

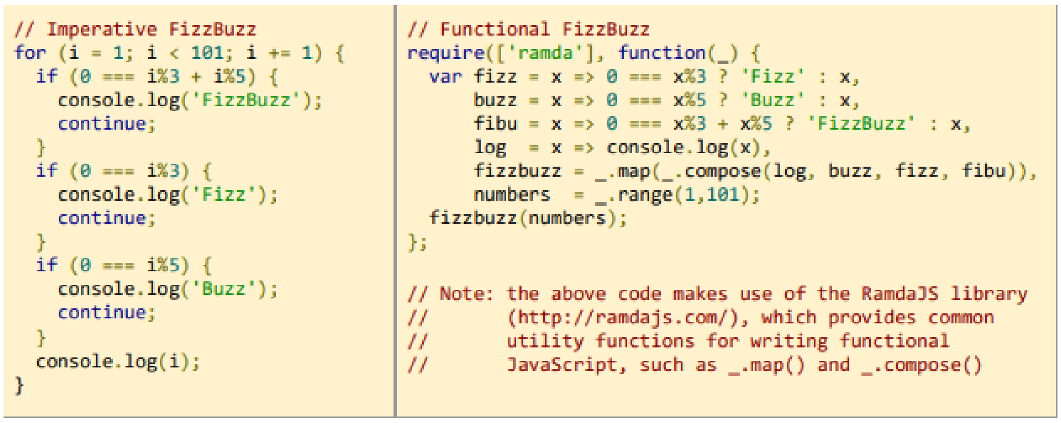
2
Related Work

Here we discuss the relevance and importance of our research and what work has been done within this subject. We discuss four topics. In the first topic (2.1) we highlight the impact of functional programming concepts on the code quality of a program. In the second topic (2.2) we research the current landscape of automated feedback tools that emphasize code quality. In the third topic (2.3) we research existing implementations of refactoring imperative loops to higher-order functions. Finally, in topic (2.4) we cover the Refactor Tutor that our research will be based on.

As for the main topic of our research, supporting students to learn to improve the code quality by refactoring imperative loops to higher-order functions, we did not find any related work that have done this already.

2.1. Functional programming concepts and code quality

Benton and Radziwill [2016] analyze the impact on code quality, in particular on the testability and reusability attributes of the system, of transitioning from imperative programming to functional programming in a multiparadigm programming language. The FizzBuzz programming exercise is used to highlight the key differences between the imperative and functional implementation.



```
// Imperative FizzBuzz
for (i = 1; i < 101; i += 1) {
  if (0 === i%3 + i%5) {
    console.log('FizzBuzz');
    continue;
  }
  if (0 === i%3) {
    console.log('Fizz');
    continue;
  }
  if (0 === i%5) {
    console.log('Buzz');
    continue;
  }
  console.log(i);
}
```

```
// Functional FizzBuzz
require(['ramda'], function(_) {
  var fizz = x => 0 === x%3 ? 'Fizz' : x,
      buzz = x => 0 === x%5 ? 'Buzz' : x,
      fibu = x => 0 === x%3 + x%5 ? 'FizzBuzz' : x,
      log  = x => console.log(x),
      fizzbuzz = _.map(_.compose(log, buzz, fizz, fibu)),
      numbers  = _.range(1,101);
  fizzbuzz(numbers);
};

// Note: the above code makes use of the RamdaJS library
//       (http://ramdajs.com/), which provides common
//       utility functions for writing functional
//       JavaScript, such as _.map() and _.compose()
```

Figure 2.1: Imperative and functional solutions to the FizzBuzz problem [Benton and Radziwill, 2016]

In the imperative implementation, all instructions are sequential and are an indivisible unit. This makes it not possible to write a test for one particular instruction without testing all instructions. The authors argue that, if there is a problem with these instructions, it may be very difficult to figure out the problem. In the functional implementation, the instructions are mapped to independent functions. These functions can eventually be composed into another function to solve the FizzBuzz problem. The authors argue that the functional implementation is a better solution since:

1. It increases testability because every instruction is an independent function that can be tested in isolation.
2. It increases reusability since the functions can be reused to compose other solutions.
3. The internal implementation of the functions is not relevant for the composed function.

Casteleyn [2019] experimented to compare the differences in maintainability metrics of the SIG-model that was introduced by Heitlager et al. [2007] of an imperative, semi-functional, and functional implementation of adding and multiplying numbers in JavaScript. The four maintainability metrics that were used:

1. Cyclomatic Complexity, the number of linearly independent paths through a program's source code.
2. Duplication, the number of duplicated lines.
3. Unit Size, the number of lines per function.
4. Lines of Code, total lines of the program's source code.

All the metrics of the semi-functional and functional implementation of adding and multiplying numbers were better than the imperative implementation, with the metrics of the functional implementation being the best. This research provides us quantitative data to further support the notion that functional programming concepts improve code quality.

2.2. Automated feedback tools & Code Quality

Keuning et al. [2018] performed a systematic literature review (SLR) of automated feedback generation for programming exercises. In this review, a total of 101 tools and 146 scientific papers were covered. Keuning et al. [2018] conclude that most of these tools do not focus on program improvements, but rather on getting a program to work. Jansen et al. [2018] and Keuning et al. [2017] further support the notion that code quality is often neglected in programming exercises.

Jansen et al. [2018] experimented to improve the code quality of students through the use of the commercial tool BetterCodeHub (BCH). BCH is a code quality analyzer that can be integrated into git without any additional installations on the students' computers. The tool was introduced into two introductory courses at the University of Amsterdam. The projects of the students were analyzed at different points in time. The experiment made use of a test group (a group that did not use the BCH tool) and a questionnaire was sent out to the students. There was a statistically significant improvement in the code quality of students' assignments over the period BCH was used. However, the authors conclude that the code quality was still quite low. This can have various reasons: it could be caused by the main focus of the course, which is not code quality, or that the students did not fully understand why the BCH feedback was useful to them. As for the test group, the sample size was too small to conclude a statistically significant improvement. Finally, from the questionnaire, students did agree that BCH helped them improving their code and mostly agree that BCH supported them in the learning process.

Keuning et al. [2017] explore the code quality of two million Java programs of novice programmers recorded in four weeks of one academic year. The authors investigated the type and frequency of code quality issues, tracked the changes that students made to their program, and checked if students were better at solving code quality issues when they have a code analysis tool installed. The findings were similar to the study of Jansen et al. [2018]: the rate of fixing code quality issues is low and the use of tools had little effect on the occurrence of code quality issues.

Keuning et al. [2021] introduce a dedicated ITS, the Refactor Tutor, for learning to improve code quality by refactoring code. The system offers exercises in which the specifications and functionality are correct, but are implemented in an inelegant manner.

The students are tasked to improve the code quality of the program. The system helps the students in two ways: students can check the progress and get hints. The hints are referred as layered feedback, in which students are presented with feedback in a layered structure, each layer below containing more detailed information of the hint.

Keuning et al. [2020] also evaluated the Refactor Tutor by teachers and students, yielding positive results. The generated hints are for the majority comparable to what teachers would suggest doing next. As for the students, they had a positive attitude towards the topic and the system. The generated hints also proved useful for students solving these exercises, although there were some observations of unclear feedback messages reported by the students.

2.3. Refactoring imperative loops to higher-order functions

Gyori et al. [2013] present a tool to refactor imperative code to functional code. Specifically, it refactors Java anonymous methods to lambda expressions and refactors imperative loops to higher-order functions. We are interested in the latter refactoring. The refactoring on imperative loops takes place if a few preconditions have been met:

P1 The enhanced for loop must be iterating over an instance of java.util.Collection to obtain an instance of java.util.Stream.

P2 the body of the for loop must not throw checked exceptions since lambda expressions cannot throw exceptions.

P3 the body of the for loop cannot have more than one reference to local variables defined outside the loop that is not final or if the initial value is never changed.

P4 the body of the for loop cannot contain any break statements since break cannot be supported by chaining operations together.

P5 the body of the for loop cannot contain more than one return statement.

P6 The body of the for loop cannot contain any continue statement since continue cannot be supported by chaining operations together.

The authors have performed the refactorings of the tool on 9 widely used open-source projects to empirically evaluate the applicability of the refactorings, the improvements on code quality, and to determine whether the tool is safe. The results were positive: a total of 46% enhanced for loops were successfully refactored, operations were chained that convey intent with finer granularity and they did not find any refactorings that changed the semantics. This research provides evidence that it is possible to refactor imperative loops to higher-order functions, albeit with a few limitations.

The tool is shipped as a static analysis tool that can be included as an extension to an IDE. This tool, however, does not learn users to refactor, but rather performs the refactorings without any additional explanation.

Casteleyn [2019] developed a solution to detect and convert imperative loops to higher-order functions in JavaScript. The solution refactors JavaScript for-of loops if a few prerequisites have been met:

P1 The for-of iterates through every element of the expression (e.g. no break statement).

P2 In the body another array must receive a new item.

P3 This push from **P2** must happen once per iteration.

The solution shows that the for-of loop can be converted to the higher-order functions *map* and *filter* by employing transformations on an Abstract Syntax Tree (A tree representation of the abstract syntactic structure of source code). However, there were a few limitations put in place due to difficulties surrounding the problem. The author argues that these limitations can be solved and also presents the readers with the solutions. The refactorings were successfully validated through the use of property-based testing. With property-based testing, the equality of the original imperative loop and the refactored higher-order function can be verified as they must provide the same output given the same input.

2.4. The Refactor Tutor

Our research builds upon the Refactor Tutor of Keuning et al. [2021]. This tutoring system has already been described in **2.2**. However, since we base our work on this tutoring system, we further explore this tutoring system in depth.

Keuning et al. [2019] research how teachers would provide students hints to improve code quality and how they would approach improving code quality. The hints are compared to the output of code quality tools and the approach of teachers is evaluated step-by-step. A total of 30 teachers are presented with three low-quality implementations. The hints given by teachers often start with a question as to why something should be improved. Compared to the output of code quality tools, the hints contain increasingly more relevant detail. The step-by-step improvements and final solutions of teachers were varied. However, from the data a general guideline could be extracted:

> *"1. remove clutter first, 2. fix errors early, keep testing along the way (even teachers make mistakes), rename to meaningful names, and do larger refactorings later one step at a time."* Keuning et al. [2019]

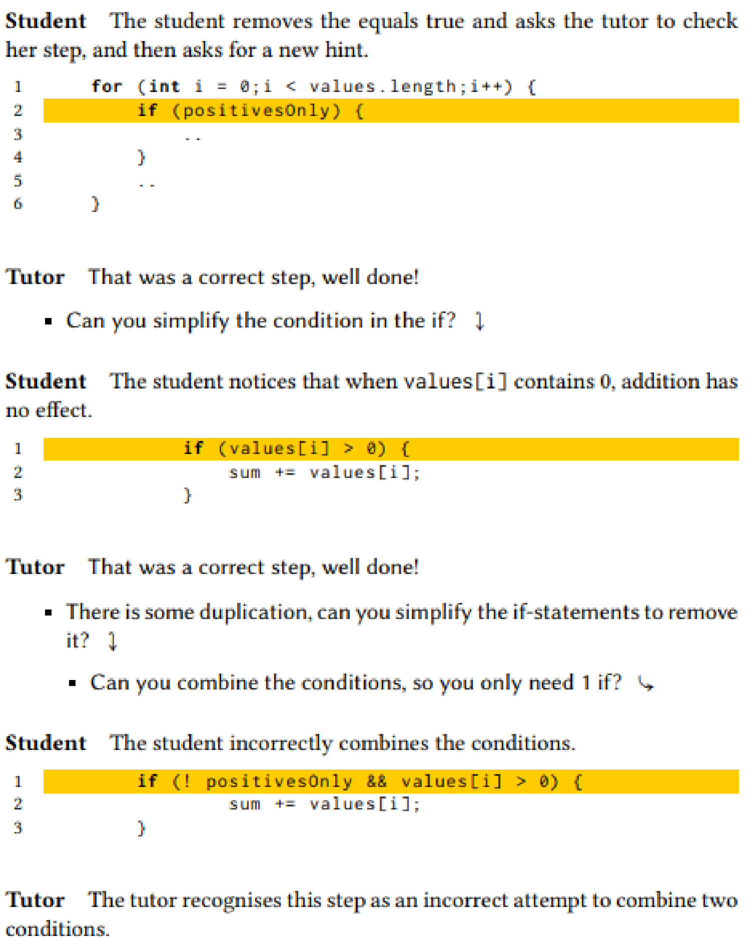
The research concludes both the *why* and *how* questions to improve code quality could be a valuable learning experience. As additional support for our research, a few teachers also suggested that imperative loops could have been rewritten as higher-order functions.

Keuning et al. [2021] start with the design of the tutoring system by asking how the system can provide feedback that is similar to the feedback of the teachers. It is then compared to other related tools and the output is thoroughly evaluated. Two examples from the research of Keuning et al. [2019] are used to identify the hints the system can provide to match the hints provided by teachers. The hints are provided in a tree structure, with lower-level hints showing more details. The hints are either presented as a question or as a solution. This matches the hints that teachers would provide, the *why* and *how*. This type of feedback is categorized by the work of Narciss [2008]. The feedback also overlaps with the categories defined by Vail and Boyer [2014] to describe the actions of human tutors when helping students learn to program. It also follows an incremental hint system similar to the work of Antonucci et al. [2015] and is seen as formative feedback since it intends to modify the thinking or behavior process of the student (defined in the work of Shute [2008]).

The implementation is built on the IDEAS [1] (Interactive domain-specific exercise assistants) framework in Haskell. This framework provides step-by-step automated hints with various feedback services. The core of the framework consists of *Rules* and *Strategies*. Rules are transformations on the data types whereas Strategies are combined as a sequence of Rules to describe the step-by-step solution to a problem. *Normalization*, transforming the program to a normal form, is used to support variants of the same program. The Refactor Tutor also uses an internal data type of a generic object-oriented language, making it easier to support multiple languages. As of writing, the Refactor Tutor currently only supports Java and consists of the following feedback services:

- The hint tree. The hint tree is the underlying data structure of all the available hints (including low-level hints) of a program in a hierarchical structure.
- The top-level hints remaining. This function calculates the current top-level hints of the program that are remaining.
- Diagnosis. The diagnosis is responsible for checking the current state of the program. This includes the parsing of the program, verifying the required functional behavior, and detecting whether rules have been (correctly) applied.

[1] Ideas - Software technology for learning and teaching. URL: `https://ideas.science.uu.nl/`. Accessed on September 3, 2021



Student The student removes the equals true and asks the tutor to check her step, and then asks for a new hint.

```
    for (int i = 0;i < values.length;i++) {
        if (positivesOnly) {
            ..
        }
        ..
    }
```

Tutor That was a correct step, well done!

- Can you simplify the condition in the if? ↓

Student The student notices that when `values[i]` contains 0, addition has no effect.

```
            if (values[i] > 0) {
                sum += values[i];
            }
```

Tutor That was a correct step, well done!

- There is some duplication, can you simplify the if-statements to remove it? ↓
  - Can you combine the conditions, so you only need 1 if? ↳

Student The student incorrectly combines the conditions.

```
        if (! positivesOnly && values[i] > 0) {
                sum += values[i];
        }
```

Tutor The tutor recognises this step as an incorrect attempt to combine two conditions.

Figure 2.2: An example of drilled-down hints with a step-by-step approach to improving the code [Keuning et al., 2021]

3
Research questions

Our research contributes towards the advancements of teaching students with an imperative programming background to use functional programming concepts. We try to do so by employing an ITS that focuses on improving the code quality.

Throughout the design of the Refactor Tutor of Keuning et al. [2021] two exercises were used as examples. We will use the first exercise to showcase our goal. The first exercise is the Sum of Values exercise with the following description:

> *"The sumValues method adds up all numbers from the array parameter, or only the positive numbers if the positivesOnly boolean parameter is set to true."* Keuning et al. [2021]

Instead of using the imperfect program as given by the system, we use the final imperative solution to highlight our refactoring to a higher-order function, which is as follows:

```
int sumValues (int[] values, boolean positivesOnly) {
    int sum = 0;
    for (int i : values) {
        if (i > 0 || !positivesOnly)
            sum += i;
    }
    return sum;
}
```

The Sum of Values exercise is a perfect example since it can be refactored to a higher-order function. The refactored implementation in Java would look as follows:

```
int sumValues(int[] values, boolean positiveOnly) {
    return Arrays.stream(values)
                .filter(x -> x > 0 || !positivesOnly)
                .sum();
}
```

In Java you must convert a collection to a Stream (Java Streams package) to be able to call the built-in higher-order functions. Note that the refactored implementation can also be further refactored with the use of of the higher-order function *reduce*. However, this is the first step towards this type of refactoring and further refactoring the functional implementation may be too advanced for students that have just begun learning a functional style.

The artifacts of this research consist of a prototype and a research thesis. The prototype extends the current feedback services *hint tree*, *top-level hints*, and *diagnosis service* of the Refactor Tutor to provide optional step-by-step feedback for refactoring certain imperative loops to higher-order functions.

3.1. Research questions

Our research is well scoped to a single problem in a specific domain. The research question of our research is formulated as:

How can we provide layered automated feedback to support students to learn to refactor imperative loops to higher-order functions?

The research question can be divided into four sub-questions. These sub-questions allow us to separate the research into four parts, which provides additional structure to our research. The answers to these sub-questions are used to provide an answer for the research question of this proposal. The sub-questions are as follows:

RQ1 *What can be found in the literature about teaching students with an imperative programming background functional programming concepts with the possible use of automated systems?*

RQ2 *How do we implement the refactorings of imperative loops to higher-order functions in the Refactor Tutor?*

RQ3 *How do we transform these refactorings to layered automated feedback?*

RQ4 *How similar is the layered automated feedback to the feedback that teachers would provide?*

The next sections of Chapter 3 discuss the research methods and validation per research question.

3.2. Teaching students with an imperative programming background functional programming concepts

Question RQ1 provides us with additional context of the current situation on how functional programming concepts are taught to students with an imperative programming background. It may also highlight the importance of our work.

Research method

To answer this question, we make use of case studies and literature reviews. The educational sector has been a subject of research for a long time now. Therefore, we apply theoretical research to find our answer. Also, given our time constraint and the remaining research questions, it is not feasible to collect primary data.

VALIDATION
The snowballing method is used as the search method for efficiently and effectively finding related literature. We start with retrieving articles by entering a combination of keywords related to the subject in scholarly search engines. Then, we inspect the citations of the articles to find other relevant literature. We continue this process by inspecting the citations of the found literature and so on. Furthermore, we will critically examine the research ourselves.

3.3. IMPLEMENTATION OF THE REFACTORINGS
Our related work shows us that it is possible to automatically refactor imperative loops to higher-order functions, albeit with a few limitations. With question RQ2, we want to implement the refactoring in the Refactor Tutor.

RESEARCH METHOD
We will use related work as our base for our refactoring. The Refactor Tutor of Keuning et al. [2021] is built on top of the IDEAS framework and contains a generic internal data type. Furthermore, the work of Gyori et al. [2013] and Casteleyn [2019] provides us with their approach and limitations to the refactoring. From thereon, we combine the related work and extend the Refactor Tutor to support our refactoring.

VALIDATION
The Refactor Tutor contains test case validation to validate that the refactored program produces the same results as the original program. With this, we can validate whether our refactoring still produces the expected results.

3.4. AUTOMATED LAYERED FEEDBACK
By answering question RQ3, we can apply our refactorings to layered feedback similar to the work of Keuning et al. [2021].

RESEARCH METHOD
To answer this question, we start with the use of examples to determine the step-by-step hints the ITS would provide the student to support our refactoring. The research of Keuning et al. [2019] provides us with general guidelines of how teachers would start improving code. After we have an initial idea, we start with developing a prototype. The prototype will extend the feedback services hint tree and top-level hints of the Refactor Tutor to provide optional step-by-step feedback for our refactoring.

VALIDATION
The feedback that will be provided must follow the same structure as the feedback given by the Refactor Tutor of Keuning et al. [2021]. In **RQ4** our feedback is compared to the feedback that teachers would provide.

3.5. AUTOMATED LAYERED FEEDBACK COMPARED TO THE TEACHERS' FEEDBACK
In question RQ3 we have extended the Refactor Tutor to support our refactoring. In question RQ4, we research the similarity between the layered automated feedback our proto-

type provides and the feedback that teachers would provide.

Research method

To answer this question, we need input from teachers. We can apply the same methods used in the research of Keuning et al. [2019] (a questionnaire). However, given our time constraint, we need to downsize the participants to at least two teachers instead. To try to handle the loss of participants, we can, instead of using a questionnaire, use a semi-structured interview. This way we can get more detailed information since we have the flexibility to ask the teachers additional questions whilst interviewing them. We can also extend the questions of the interview by asking teachers multiple approaches to refactoring the program.

Validation

The teachers that will be questioned must be teaching in a higher education environment and must teach computer science or a similar study for at least 2 years. As seen in the research of Keuning et al. [2019], the teachers' approaches to improving code are varied. Therefore, we interview at least two teachers. Furthermore, the questions will be documented and the interviews with the teachers will be recorded and transcribed to document the answers of the teachers.

4 Teaching students with an imperative programming background functional programming concepts

This chapter relates to research question 1: What can be found in the literature about teaching students with an imperative programming background functional programming concepts with the possible use of automated systems?

There is a lot of existing and ongoing research on teaching students functional programming concepts. Section 4.1 discusses a few approaches of teaching functional programming concepts. These approaches do not leverage the imperative programming background of students. Therefore, Section 4.2 discusses another domain, called multiparadigm programming, that combines multiple programming paradigms with each other.

4.1. Teaching functional programming

Joosten et al. [1993] introduce a problem-solving method for teaching functional programming to students. The authors mention that some students with considerable experience of programming in an imperative language will need to force themselves to look afresh at the process of programming. Given the problem-solving method (which is based on an introductory investigation of the mathematical method), the authors argue that students are equipped with the tools to enable them to write complex programs in disciplined ways.

Kristensen et al. [2001] research the impact of teaching object-oriented programming after functional programming. The authors saw a significant improvement in the produced object-oriented code as well as in the program documentation and argue that this is due to the high-level abstraction of what functional programming provides.

Chakravarty and Keller [2004] argue that we should not focus on teaching first-year students purely functional programming nor procedural, object-oriented, or logic programming, but rather focus on teaching general concepts. However, the authors argue that teaching general concepts can best be done by using purely functional programming languages due to the light syntax and clean semantics, and the high level of abstraction.

Gerdes et al. [2016] introduce an Intelligent Tutoring System, AskElle, for teaching students functional programming. AskElle and the Refactor Tutor are both built upon the IDEAS framework so they share the same set of functionalities. However, the goals differ from each other. The goal of AskElle is to teach functional programming in Haskell whereas the goal of the Refactor Tutor is to teach code improvements in the imperative programming language Java.

4.2. Multiparadigm programming

Ross [1998] experimented with teaching functional programming concepts by applying them at an early stage to imperative programming to improve the students' understanding of programming paradigms and to facilitate an easier transition to functional programming. The authors developed a programming style that is based on a functional style. The students were asked to use this style throughout a course using an imperative programming language. After this course, the students went on with a further course teaching functional programming. 90% of the students said that the two courses made them more aware of different programming paradigms. Only 21% felt most home with a functional paradigm, with 60% that felt most home with imperative. Furthermore, 47% of students were comfortable with the concept of higher-order functions. According to the authors, this indicates how strongly the imperative mindset has been internalized. The authors finally concluded that taking functional paradigm ideas into imperative language programming certainly has helped in learning both LISP and understanding the functional paradigm.

Furthermore, we see an emergence of multiple programming paradigms being combined into a single programming language. With Budd [1994] introducing a multiparadigm programming language Leda, Odersky et al. [2008] creating the object-oriented functional programming language Scala and with modern and older programming languages adapting multiple programming paradigms. For example, the work of Ortin et al. [2017] uses C#, inherently an imperative programming language, to teach students object-oriented, functional, concurrent, parallel, and meta-programming.

However, with the systematic literature review of Keuning et al. [2018] on tutoring systems and our research, we have not found a multiparadigm programming tutoring system that focuses on teaching functional programming concepts by leveraging the imperative programming background of students (and with the intent of code quality improvement). We believe that this further highlights the importance of our work.

5
Implementation of the refactorings

This chapter relates to research question 2: How do we implement the refactorings of imperative loops to higher-order functions in the Refactor Tutor?

The Refactor Tutor only provides imperative code feedback. To support our refactoring to higher-order functions, we determined that two things were missing:

1. Support for lambda expressions in the internal data types.
2. Support for (higher-order) functions.

5.1. Supporting lambda expressions in the internal data types

There are two internal data types in the Refactor Tutor: the Syntax and AST data types. The Syntax data type is larger than the AST data type and is used for complex calculations such as the calculation of program improvements. The AST data type is a simplified version and is used for smaller calculations where speed is often of importance (the evaluation of the program on its input and outputs). Furthermore, It is easier to cover cases as we work with a smaller data type. For example, the Syntax data type has three types of loops (While, For, For Each) whereas the AST data type only has one type of loop (While). It is possible to convert the Syntax and AST data types back and forth. However, when converting the AST back to a Syntax, specific information may be lost as it is not stored in the AST. Lastly, the Java code is only parsed to the Syntax data type.

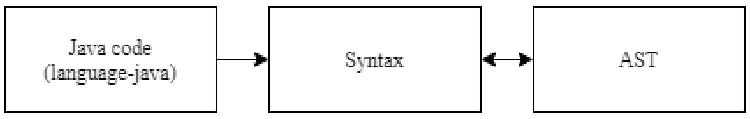
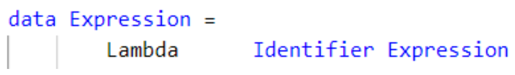


Figure 5.1: Conversion of the data types in the Refactor Tutor

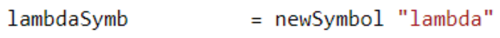
For the Syntax data type we need to extend the Expression type to support Lambda expressions. The Expression type consists of variables, operators, literals and method calls. We included a Lambda constructor in the Expression type with two fields: Identifier and Expression. The Identifier references the lambda parameter and the Expression holds the expression of the lambda expression. For now, we only support Lambda expressions with a single parameter to keep it simple.



```haskell
data Expression =
    |   Lambda      Identifier Expression
```

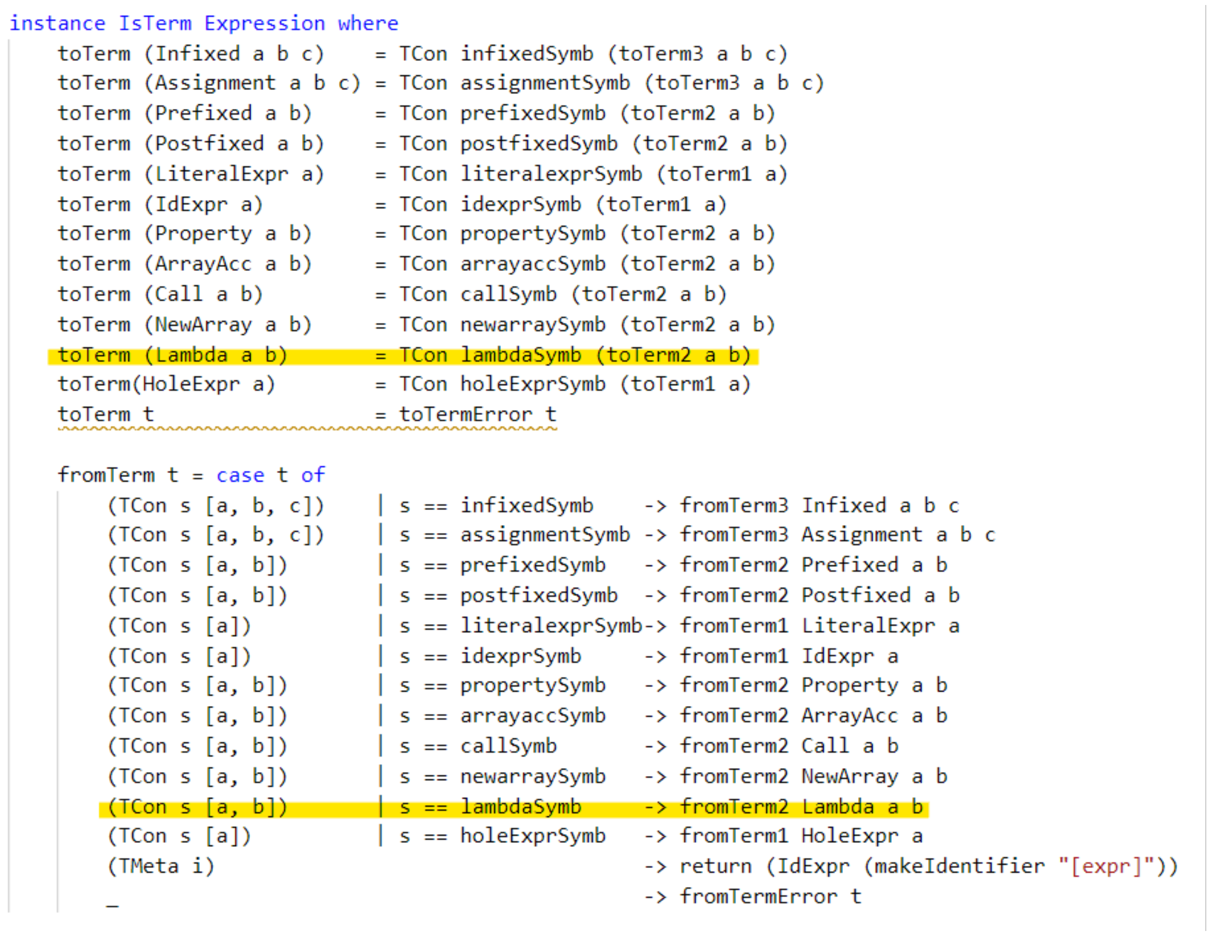
Figure 5.2: Lambda constructor in the Expression type of the Syntax data type

Furthermore, for every Expression constructor we need to add a Term implementation. This is required by the IDEAS framework to traverse sub-expressions. The IDEAS framework offers a pre-defined method of implementing a Term: A symbol must be defined for the constructor, the symbol is then used in the IsTerm instance to convert from and to a Term with the functions fromTerm and toTerm.



```haskell
lambdaSymb          = newSymbol "lambda"
```

Figure 5.3: A Symbol is defined for the Lambda constructor



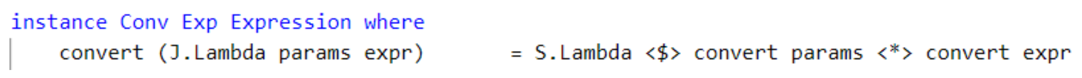
```haskell
instance IsTerm Expression where
    toTerm (Infixed a b c)    = TCon infixedSymb (toTerm3 a b c)
    toTerm (Assignment a b c) = TCon assignmentSymb (toTerm3 a b c)
    toTerm (Prefixed a b)     = TCon prefixedSymb (toTerm2 a b)
    toTerm (Postfixed a b)    = TCon postfixedSymb (toTerm2 a b)
    toTerm (LiteralExpr a)    = TCon literalexprSymb (toTerm1 a)
    toTerm (IdExpr a)         = TCon idexprSymb (toTerm1 a)
    toTerm (Property a b)     = TCon propertySymb (toTerm2 a b)
    toTerm (ArrayAcc a b)     = TCon arrayaccSymb (toTerm2 a b)
    toTerm (Call a b)         = TCon callSymb (toTerm2 a b)
    toTerm (NewArray a b)     = TCon newarraySymb (toTerm2 a b)
    toTerm (Lambda a b)       = TCon lambdaSymb (toTerm2 a b)
    toTerm(HoleExpr a)        = TCon holeExprSymb (toTerm1 a)
    toTerm t                  = toTermError t

    fromTerm t = case t of
        (TCon s [a, b, c])    | s == infixedSymb    -> fromTerm3 Infixed a b c
        (TCon s [a, b, c])    | s == assignmentSymb -> fromTerm3 Assignment a b c
        (TCon s [a, b])       | s == prefixedSymb   -> fromTerm2 Prefixed a b
        (TCon s [a, b])       | s == postfixedSymb  -> fromTerm2 Postfixed a b
        (TCon s [a])          | s == literalexprSymb-> fromTerm1 LiteralExpr a
        (TCon s [a])          | s == idexprSymb     -> fromTerm1 IdExpr a
        (TCon s [a, b])       | s == propertySymb   -> fromTerm2 Property a b
        (TCon s [a, b])       | s == arrayaccSymb   -> fromTerm2 ArrayAcc a b
        (TCon s [a, b])       | s == callSymb       -> fromTerm2 Call a b
        (TCon s [a, b])       | s == newarraySymb   -> fromTerm2 NewArray a b
        (TCon s [a, b])       | s == lambdaSymb     -> fromTerm2 Lambda a b
        (TCon s [a])          | s == holeExprSymb   -> fromTerm1 HoleExpr a
        (TMeta i)                                   -> return (IdExpr (makeIdentifier "[expr]"))
        _                                           -> fromTermError t
```

Figure 5.4: The fromTerm and toTerm implementation of the Lambda constructor in the IsTerm instance are highlighted

As for the AST, we want to keep it as small as possible. Therefore, the higher-order functions and lambda expressions of the Syntax data type are converted to a While loop of the AST data type. In the next section we discuss how we have implemented this.

To parse the Java code, the Refactor Tutor uses the language-java Haskell package. This package supports Java 8, thus supporting lambda expressions. We extended the internal Java parser that is responsible for converting the language-java data types to the Syntax data types to support lambda expressions.



```
instance Conv Exp Expression where
    convert (J.Lambda params expr)       = S.Lambda <$> convert params <*> convert expr
```

Figure 5.5: Conversion of the Java Lambda expression to the Syntax Lambda expression

5.2. Support for (higher-order) functions

As mentioned in the work of Casteleyn [2019], the *map*, *filter*, and *reduce* are the most used higher-order functions. For our prototype, we support *map* and *filter* since these functions are researched in the work of Casteleyn [2019] and we want to support at least two higher-order functions to showcase the differences.

These functions need to be called on collections, and in the case of Java, on Java Streams. The Refactor Tutor only supports arrays due to a known bug on collections. An array is a specific implementation of a collection. Therefore, we continue the refactoring on arrays. Furthermore, the Refactor Tutor does not support Java Streams. We omit Java Streams from the refactoring as it is not necessary to highlight our goal (refactoring to a higher-order function). We can support collections and Java Streams in the refactoring when next versions of the Refactor Tutor support it as well. Our program will not be compilable due to dropping support of Java Streams, but it still produces valid Java syntax. This does not cause any problems since the Refactor Tutor does not compile the Java program, but uses a Java syntax parser to convert to its internal data type.

Method calls are partially supported in both the Syntax and AST data types. The method calls are stored in a type, but the behavior of the methods are not stored. For example, the *add* method call may be stored in a type, but the Refactor Tutor does not know what this method does. The AST data type is used for the evaluation of a program on its inputs and outputs. To support a higher-order function in the evaluation of a program, the Refactor Tutor needs to know what a higher-order function does. To solve this problem, we convert a Syntax higher-order function *map* and *filter* to an AST While loop. This is done by modifying the AST parser, the parser responsible for parsing the Syntax data type to the AST data type.

Additionally, the Refactor Tutor also needs to support the behavior of the method call *add* for the imperative implementation and the converted AST While loop (from a Syntax higher-order function) in the AST data type. It is difficult to convert the Syntax representation (either a type of for loop containing in its body the method call *add* or a higher-order function) directly to an AST representation since we would need to convert it to an AST array. To resolve the issue in a more simple way, we extended the Evaluator (responsible for the evaluation of the program) to perform an add action (adds the parameter of the method to the designated array) if the evaluator matches with a method call *add*.

5.3. Example exercises

We created two example exercises to showcase our program transformation to a *map* and *filter*. The Refactor Tutor has built-in test case validation for exercises. This provides us additional validation that the refactored program produces the same output as before.

The squareValues imperative implementation and higher-order implementation of *map* is as follows:

```
public static int[] squareValues(int[] values) {
    int[] squaredValues = new int[0];

    for(int value : values) {
        squaredValues.add(value * value);
    }

    return squaredValues;
}

public static int[] squareValues(int[] values) {
   return values.map(value -> value * value);
}
```

The squareValues method has the following test cases defined in the Refactor Tutor:

```
    {1, 2, 3, 4, 5} > {1, 4, 9, 16, 25}
    {-5, -4, -3, -2, -1} > {25, 16, 9, 4, 1}
    {0} > {0}
    {} > {}
```

The left side represents the input of the program and on the right side the expected output.

The filterOnlyPositiveValues imperative implementation and higher-order implementation of *filter* is as follows:

```
public static int[] filterOnlyPositiveValues(int[] values) {
    int[] positiveValues = new int[0];

    for(int value : values) {
        if(value > 0) {
            positiveValues.add(value);
        }
    }

    return positiveValues;
}

public static int[] filterOnlyPositiveValues(int[] values) {
    return values.filter(value -> value > 0);
}
```

The filterOnlyPositiveValues method has the following test cases defined in the Refactor Tutor:

```
{1, 2, 3, 4, 5} > {1, 2, 3, 4, 5}
{-5, -4, -3, -2, -1} > {}
{0} > {}
{} > {}
```

These example exercises will satisfy the pre-conditions, constraints, and rules of our program transformation. This will ensure that the hint to refactor to a *map* or *filter* is always shown. To recognize more examples, the pre-conditions, rules, and transformations would need to be modified to allow more flexibility for matching imperative code to be refactored to a higher-order function.

5.4. Program Transformation

We now have everything in order to refactor an imperative loop to a higher-order function *map* or *filter*. In this section, we introduce the program transformations of transforming certain for each loops to our supported higher-order functions. The transformations will be performed if certain pre-conditions and rules have been satisfied.

5.4.1. Pre-conditions and rules

Given the context of the Refactor Tutor (a controlled learning environment), we do not need to support complex cases since we do not perform real-world refactorings and are in an environment where we define our exercises for students to learn. For our prototype, we try to keep things as simple as possible by not trying to recognize all cases where a loop can be refactored to a higher-order function.

We use the pre-conditions of Gyori et al. [2013] and adjust it into our context:

P1 The enhanced for loop must be iterating over an instance of java.util.Collection to obtain an instance of java.util.Stream.

P2 the body of the for loop must not throw checked exceptions since lambda expressions cannot throw exceptions.

P3 the body of the for loop cannot have more than one reference to local variables defined outside the loop that is not final or if the initial value is never changed.

P4 the body of the for loop cannot contain any break statements since break cannot be supported by chaining operations together.

P5 the body of the for loop cannot contain more than one return statement.

P6 The body of the for loop cannot contain any continue statement since continue cannot be supported by chaining operations together.

P1 is modified to an instance of array. **P2** is not applicable since the Refactor Tutor does not support exceptions. **P3** is omitted for simplification purposes and is applied as a constraint in our exercises. At last, **P5** is modified to no return statement. The LambdaFicator uses the properties anyMatch and noneMatch to refactor some loops that contain return statements. For simplification purposes, we have decided to modify this pre-condition to no return statements.

If the conditions modified **P1**, **P4**, modified **P5** and **P6** have been satisfied, the next step is to determine whether the loop can be refactored to the higher-order function *map* or *filter*. We use the same approach as Casteleyn [2019] to determine this: by employing an AST and defining rules that the tree must match. We have kept the rules simple, but strict. To match a loop that can be refactored to either a *map* or *filter*, the loop must follow the exact rules:

1. The loop must be a for each. All loops can be refactored to a for each, as seen in the work of Casteleyn [2019]. Oftentimes, the Refactor Tutor also provides hints to refactor to the for each loop.

2. (Only for *filter*) The loop must only contain an if body.

3. In the body of the loop (or if) there must only be one method call *add* on a defined array with its parameter the item in the for each loop.

These rules can be extended and modified for additional flexibility. If these rules are all matched, it then means the transformation can take place.

5.4.2. Transformation

To transform the for each loop to a higher-order function, we transform the Syntax For Each type (Syntax representation of the imperative implementation) to a Syntax Call with a Lambda Expression (Syntax representation of a higher-order function). Eventually, the Syntax representations will be transformed to an AST While loop. The transformation for the example exercise *filter* (which is an expansion of *map*) looks as follows:

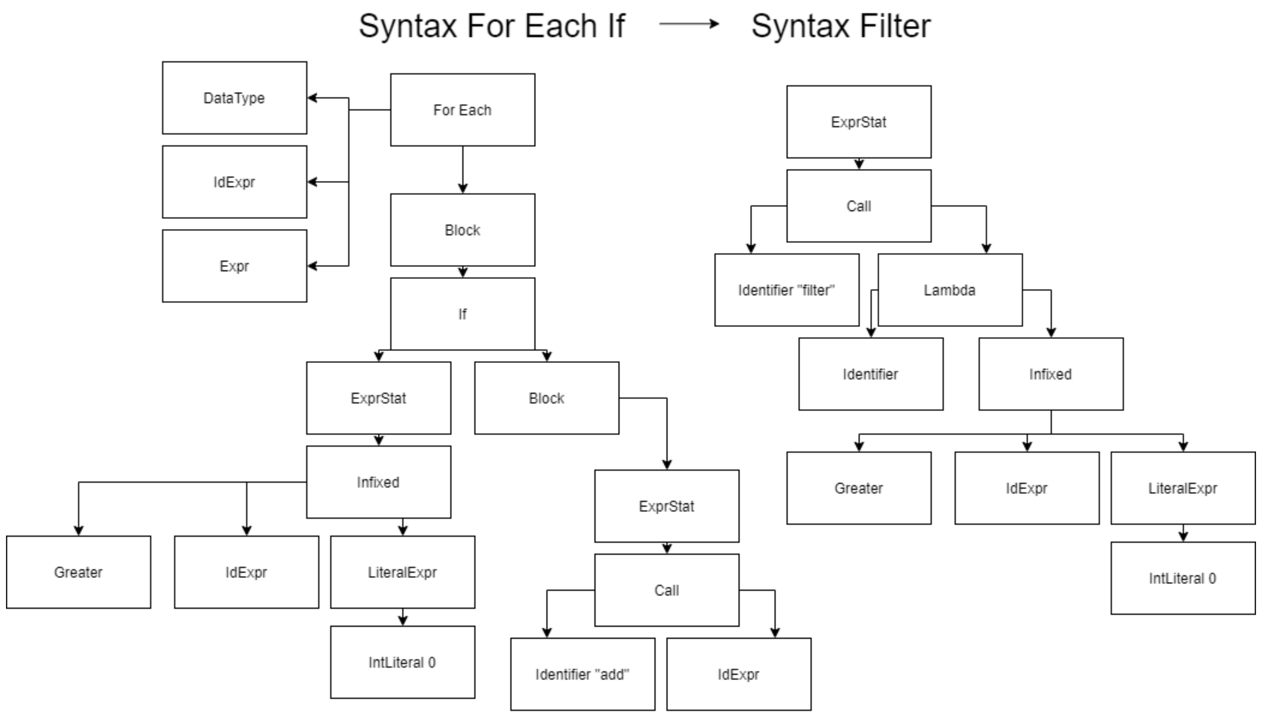


Figure 5.6: Syntax For Each If transformed to the Syntax Filter. Small details are shown/hidden based on relevancy

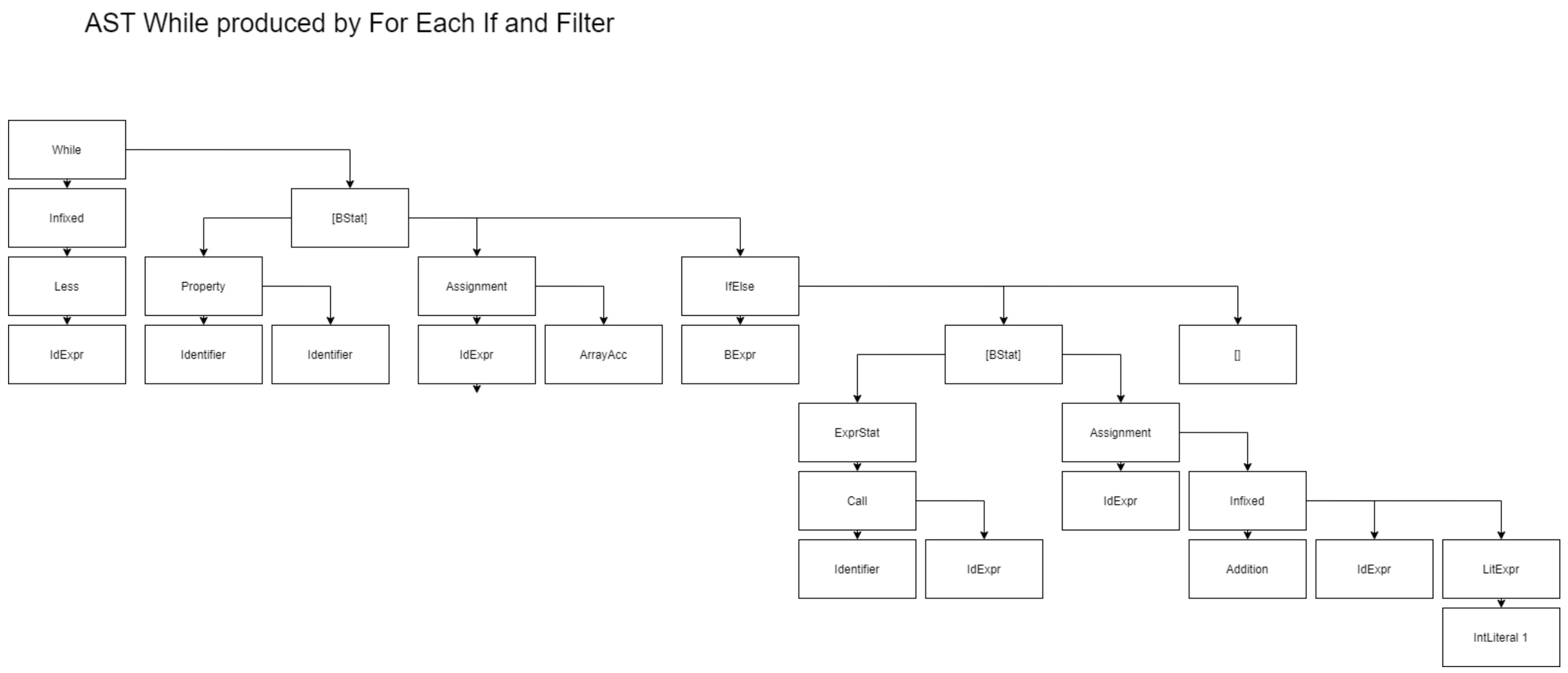


Figure 5.7: AST While loop produced both by the Syntax For Each If and Filter. Small details are shown/hidden based on relevancy

The Syntax For Each with If and the Syntax Filter produce the same AST While data structure. With this, we conclude that the imperative implementation with the Syntax For Each with If is equal to the Syntax Filter when converting to the AST data type.

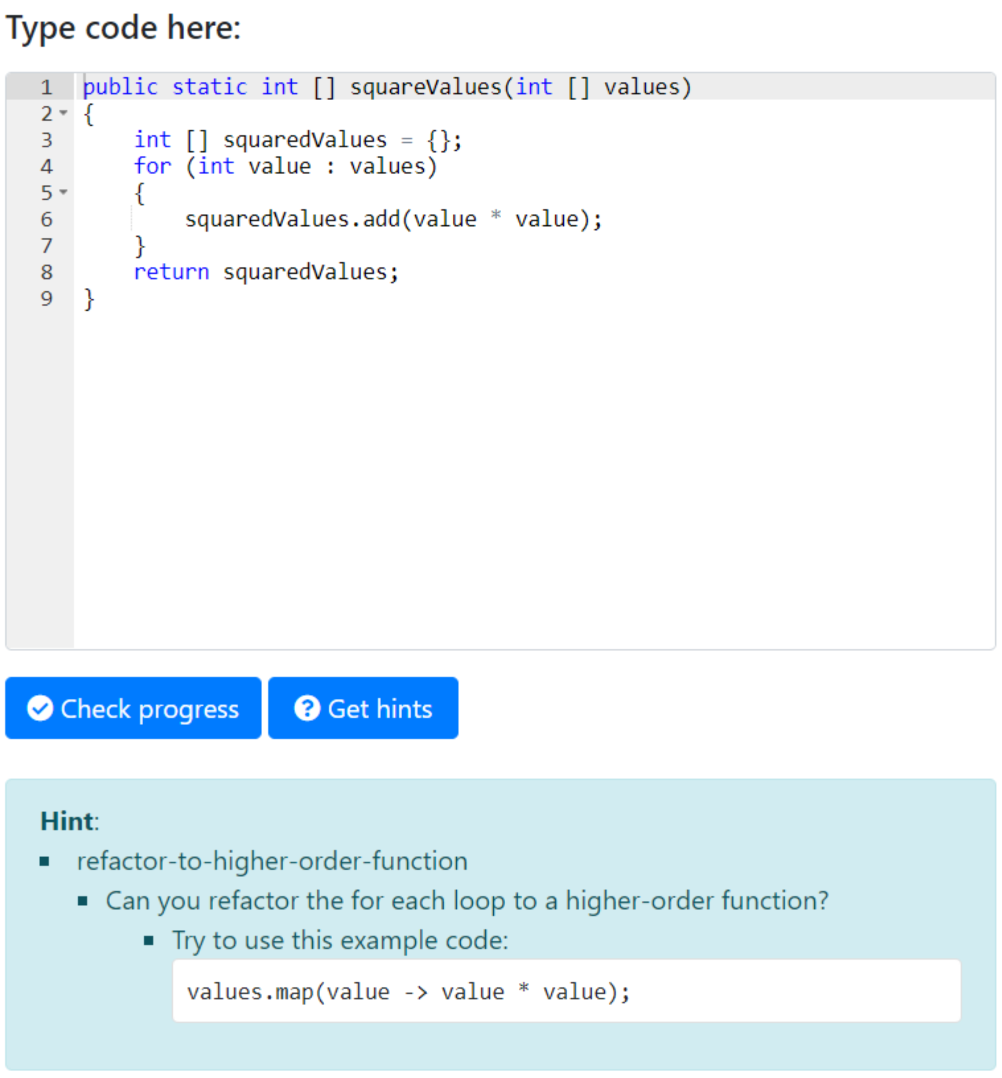
5.5. Hint System

The program transformation must be integrated into the hint system of the IDEAS framework to represent it as a single hint. Our program transformation follows the same structure as other program transformations, allowing us to easily adapt the program transformation into the hint system. First, we need to create a Rule type for our program transformation. A Rule in the Refactor Tutor can be seen as a hint since a hint is a rule with an additional feedback description. The constructor of the Rule type has the fields identifier and a generic value. Our rule has the identifier *forEachToHigherOrderFunction* and as value a Context type with a Syntax Statement type. The Context type stores the environment, is used for traversing the Syntax Statement type, and must be used for further usage of the IDEAS framework. The Syntax Statement type represents a programming language instruction (e.g. If, ForEach, Break) which can also contain another programming language instruction.

We need to adapt the Rule into the IDEAS Strategy type. The Strategy type is used to combine all rules, determine the order of the rules, and when the rules should occur through traversal. We created a Strategy type to prevent interference from the existing Refactor Tutor Strategy and its rules. The Strategy type is also the expected type for the IDEAS Exercise type. The Exercise type contains all the components necessary for calculating feedback for one class of exercises. In the Exercise type of the Refactor Tutor we modified the Strategy to our Strategy. The Refactor Tutor now uses our Strategy.

5.6. Showcasing refactoring in the Refactor Tutor UI

The Refactor Tutor UI with the hints of refactoring to a *map* and *filter* looks as follows:

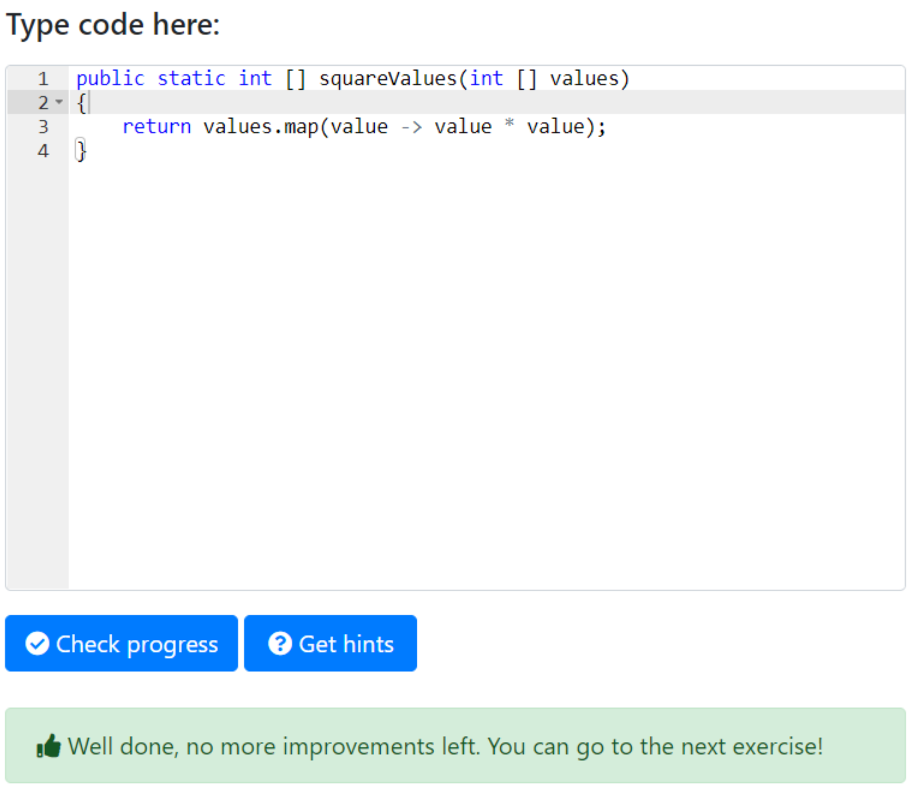

Type code here:

```
public static int [] squareValues(int [] values)
{
    int [] squaredValues = {};
    for (int value : values)
    {
        squaredValues.add(value * value);
    }
    return squaredValues;
}
```

Hint:
- refactor-to-higher-order-function
  - Can you refactor the for each loop to a higher-order function?
    - Try to use this example code:

```
values.map(value -> value * value);
```

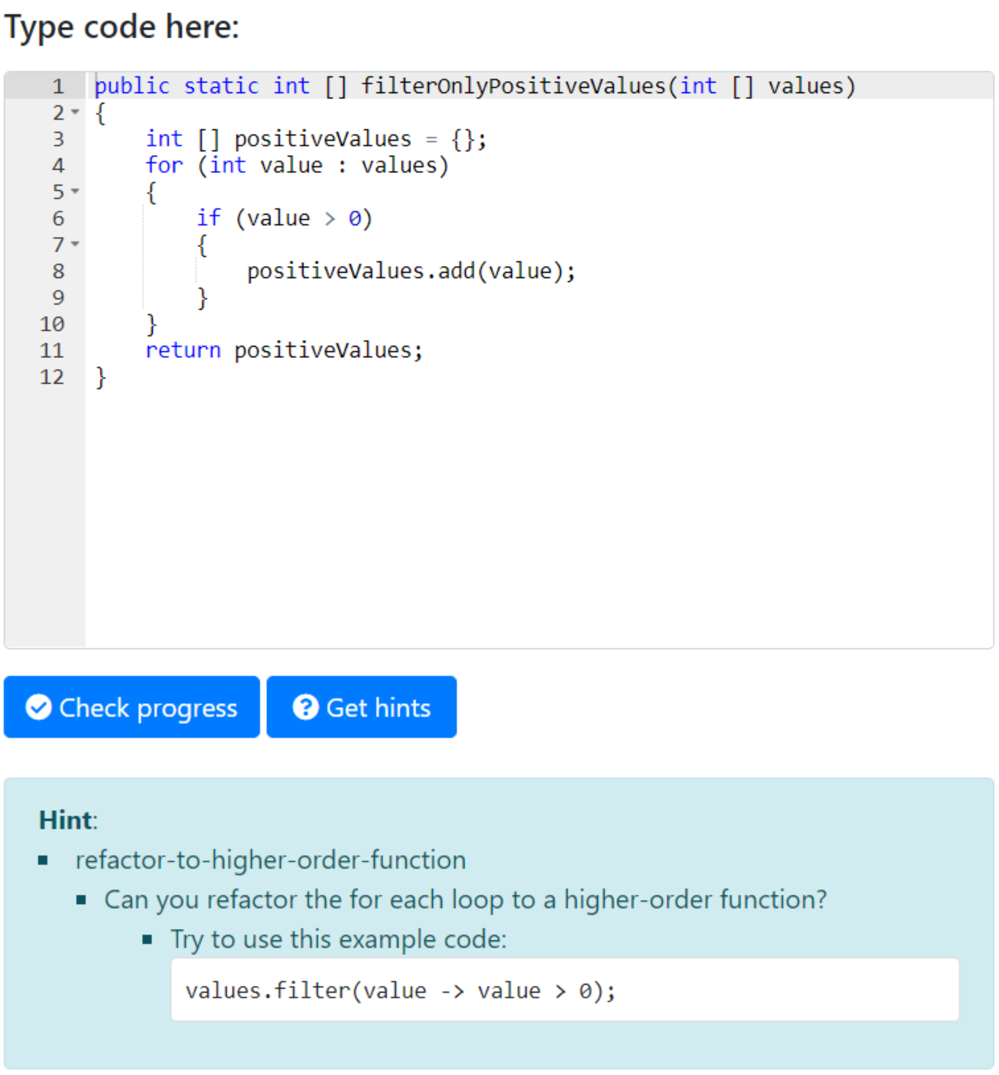
Figure 5.8: Refactor UI - Hint to refactor the imperative implementation of the *squareValues* method to the higher-order function *map*



```
public static int [] squareValues(int [] values)
{
    return values.map(value -> value * value);
}
```

Well done, no more improvements left. You can go to the next exercise!

Figure 5.9: Refactor UI - The refactored higher-order function *map* implementation of the *squareValues* method


Type code here:

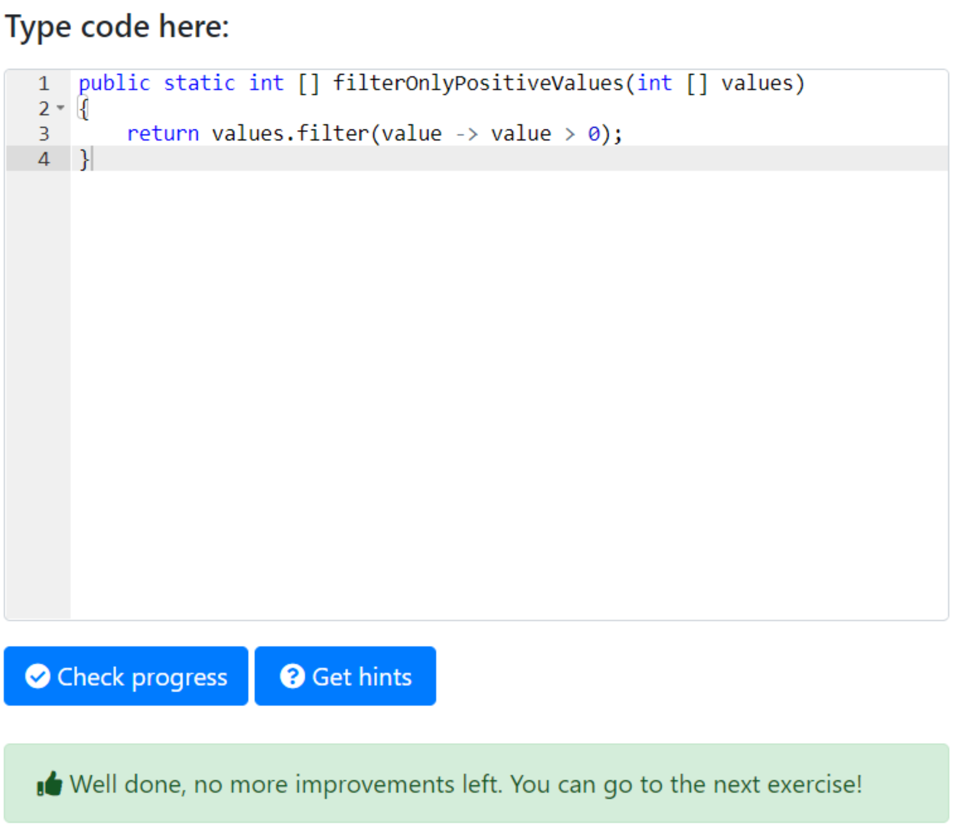
```
public static int [] filterOnlyPositiveValues(int [] values)
{
    int [] positiveValues = {};
    for (int value : values)
    {
        if (value > 0)
        {
            positiveValues.add(value);
        }
    }
    return positiveValues;
}
```

Hint:

- refactor-to-higher-order-function
  - Can you refactor the for each loop to a higher-order function?
    - Try to use this example code:

```
values.filter(value -> value > 0);
```

Figure 5.10: Refactor UI - Hint to refactor the imperative implementation of the *positiveValues* method to the higher-order function *filter*


Type code here:

```
public static int [] filterOnlyPositiveValues(int [] values)
{
    return values.filter(value -> value > 0);
}
```

Well done, no more improvements left. You can go to the next exercise!

Figure 5.11: Refactor UI - The refactored higher-order function *filter* implementation of the *positiveValues* method

6 AUTOMATED LAYERED FEEDBACK OF THE REFACTORINGS

This chapter relates to research question 3: How do we transform these refactorings to layered automated feedback?

In the previous chapter, we have implemented the refactorings in the Refactor Tutor, but the refactorings are provided through a single hint. This does not provide the students with a step-by-step walk-through and does not teach the *why* and *how* of the refactorings. It provides similar feedback as static analysis tools.

In this chapter, we analyze the process of the refactorings and try to provide the students with step-by-step hints to help them understand the refactoring.

6.1. REFACTORING

First, we investigate what exactly needs to be refactored. We use the two exercises introduced in Chapter 5 to showcase the *map* and *filter* higher-order functions.

6.1.1. MAP

The imperative implementation of the square values method is as follows:

```
public static int[] squareValues(int[] values) {
    int[] squaredValues = new int[0];

    for(int value : values) {
        squaredValues.add(value * value);
    }

    return squaredValues;
}
```

and can be refactored to:

```
public static int[] squareValues(int[] values) {
    return values.map(value -> value * value);
}
```

Only the argument of the add method (*value * value*) remains in both implementations. As we can see from this example, the imperative implementation is completely different from the refactored implementation. However, we would still need to provide the student with step-by-step hints to understand the refactoring.

6.1.2. Filter

The imperative implementation of the filter only positive values method is as follows:

```
public static int[] filterOnlyPositiveValues(int[] values) {
    int[] positiveValues = new int[0];

    for(int value : values) {
        if(value > 0) {
            positiveValues.add(value);
        }
    }

    return positiveValues;
}
```

and can be refactored to:

```
public static int[] filterOnlyPositiveValues(int[] values) {
    return values.filter(value -> value > 0);
}
```

Only the if expression (*value* > *0*) remains in both implementations. In this example, we see that the imperative implementation is completely different from the refactored implementation.

6.2. Step-by-step feedback

The steps that we provide need to guide the student to understandably refactor an imperative loop to a higher-order function. Here, we discuss the steps for the refactoring.

6.2.1. Type of feedback

We used the general guidelines provided by the work of Keuning et al. [2019] as a base for setting up our steps.

> *"1. remove clutter first, 2. fix errors early, keep testing along the way (even teachers make mistakes), rename to meaningful names, and do larger refactorings later one step at a time."* Keuning et al. [2019]

Our refactoring is a large refactoring where we remove almost all the code and replace it with a one-liner. We start with removing clutter first. Where possible, we re-use pieces of code and rename them. The student is presented with each step at a time. The student can verify if the step is successfully completed. Afterward, the student is presented with the next step until the student reaches the last step. Since we provide the steps each at a time and validate each step separately, the student can fix the mistakes at the earliest. When the student finishes the last step, we can validate the end solution with the input/output testing.

Furthermore, our exercises will start with a small introduction since we introduce new concepts which require additional information before starting the step-by-step refactoring. It may be that this type of feedback may not be the best fit for our refactoring. However, we will evaluate our feedback against the feedback that teachers would provide.

6.2.2. Steps for map

For refactoring the imperative implementation to the higher-order function map, we have defined the following steps with the whys and hows:

1. We start with a small introduction of higher-order functions and in particular the map function. From the introduction, it must be clear that:

    - The higher-order function goes through each element of the collection
    - The higher-order function map maps each element of the collection to another element
    - The higher-order function map returns a new collection

    ```
    public static int[] squareValues(int[] values) {
        int[] squaredValues = new int[0];

        for(int value : values) {
            squaredValues.add(value * value);
        }

        return squaredValues;
    }
    ```

2. Remove the instantiation of the *squaredValues* array. Why? It is not needed anymore since the higher-order function map will return a new array. How? Removing the *squaredValues* array instantiation.

    ```
    public static int[] squareValues(int[] values) {
        for(int value : values) {
            squaredValues.add(value * value);
        }

        return squaredValues;
    }
    ```

3. Remove the for loop. Why? It is not needed anymore since the higher-order function map goes through each element of the collection. How? Removing the for loop excluding the body.

```
    public static int[] squareValues(int[] values) {
        squaredValues.add(value * value);

        return squaredValues;
    }
```

4. Remove the *squaredValues* variable from the return statement. Why? The variable does not exist anymore. How? Removing the variable from the return statement.

```
    public static int[] squareValues(int[] values) {
        squaredValues.add(value * value);

        return;
    }
```

5. Rename the *squaredValues* (*squaredValues.add(value * value)*) to *values.* Why? We will apply the map function on the original (*values*) array as it will produce a new array. How? Renaming the *squaredValues* to *values.*

```
    public static int[] squareValues(int[] values) {
        values.add(value * value);

        return;
    }
```

6. Rename the method *add* to *map*. Why? We do not need to add an element to the array anymore, but the higher-order function map does this for us. How? Renaming the *add* method call to *map.*

```
    public static int[] squareValues(int[] values) {
        values.map(value * value);

        return;
    }
```

7. Replace the argument of the map method *value * value* to *value -> ?*. In this step, we introduce some information on lambdas. From this introduction, it must be clear that:

   - *value* represents a single item of the collection and can be given any name. It is the parameter of the lambda expression.
   - -> is the lambda operator that separates the left side (parameter) and the right side of the lambda expression.

We also let the student know that the ? is a placeholder which will be filled in later on. Why? Introducing step-by-step lambdas and how they work with higher-order functions. How? Adding to the parameter of the map method the following *-> ?*.

```
public static int[] squareValues(int[] values) {
    values.map(value -> ?);

    return;
}
```

8. Replace the *?* with *value * value*. In this step, we introduce additional information on lambdas. From this information, it must be clear that:

   - On the right side of the lambda expression (currently filled in with *?*) we return a value for the specific item in the parameter.

   Why? We have completed the full lambda expression. How? Replace the ? with *value * value*.

```
public static int[] squareValues(int[] values) {
    values.map(value -> value * value);

    return;
}
```

9. Place the *values.map(value -> value * value)* directly into the return statement. Why? We can return the result of the map directly to the return of the method. How? Placing the statement after the return.

```
public static int[] squareValues(int[] values) {
    return values.map(value -> value * value);
}
```

6.2.3. Steps for filter

For refactoring the imperative implementation to the higher-order function map, we have defined the following steps with the whys and hows:

1. We start with a small introduction of higher-order functions and in particular the filter function. From the introduction, it must be clear that:

   - A higher-order function goes through each element of the collection
   - The higher-order function filter filters each element of the collection based on a condition
   - The higher-order function filter returns a new collection

```
public static int[] filterOnlyPositiveValues(int[] values) {
    int[] positiveValues = new int[0];

    for(int value : values) {
        if(value > 0) {
            positiveValues.add(value);
        }
    }

    return positiveValues;
}
```

2. Remove the instantiation of the *positiveValues* array. Why? It is not needed anymore since the higher-order function filter will return a new array. How? Removing the *positiveValues* array instantiation.

```
public static int[] filterOnlyPositiveValues(int[] values) {
    for(int value : values) {
        if(value > 0) {
            positiveValues.add(value);
        }
    }

    return positiveValues;
}
```

3. Remove the for loop. Why? It is not needed anymore since the higher-order function filter goes through each element of the collection. How? Removing the for loop excluding the body.

```
public static int[] filterOnlyPositiveValues(int[] values) {
    if(value > 0) {
        positiveValues.add(value);
    }

    return positiveValues;
}
```

4. Remove the if statement. Why? Because it is not needed anymore since the check will be performed in the higher-order function method. How? Remove the if excluding the body.

```
public static int[] filterOnlyPositiveValues(int[] values) {
    positiveValues.add(value);

    return positiveValues;
}
```

5. Remove the *positiveValues* variable from the return statement. Why? The variable does not exist anymore. How? Removing the variable from the return statement.

   ```
   public static int[] filterOnlyPositiveValues(int[] values) {
       positiveValues.add(value);

       return;
   }
   ```

6. Rename the *positiveValues* (*positiveValues.add(value)*) to *values*. Why? We will apply the filter function on the original (*values*) array as it will produce a new array. How? Renaming the *positiveValues* to *values*.

   ```
   public static int[] filterOnlyPositiveValues(int[] values) {
       values.add(value);

       return;
   }
   ```

7. Rename the method *add* to *filter*. Why? We do not need to add positive values to an array anymore, but the higher-order function filter will filter out the non-positive values. How? Renaming the *add* method call to *filter*.

   ```
   public static int[] filterOnlyPositiveValues(int[] values) {
       values.filter(value);

       return;
   }
   ```

8. Extend the argument of the filter method *value* to *value -> ?*. In this step, we introduce some information on lambdas. From this introduction, it must be clear that:

   - *value* represents a single item of the collection and can be given any name. It is the parameter of the lambda expression.
   - -> is the lambda operator that separates the left side (parameter) and the right side of the lambda expression.

   We also let the student know that the ? is a placeholder which will be filled in later on. Why? Introducing step-by-step lambdas and how they work with higher-order functions. How? Adding to the parameter of the filter method the following *-> ?*.

   ```
   public static int[] filterOnlyPositiveValues(int[] values) {
       values.filter(value -> ?);

       return;
   }
   ```

9. Replace the *?* with *value > 0*. In this step, we introduce additional information on lambdas. From this information, it must be clear that:

   - On the right side of the lambda expression (currently filled in with *?*) we return the condition to filter out the non-positive values

   Why? We have completed the full lambda expression. How? Replace the ? with *value > 0*.

```
public static int[] filterOnlyPositiveValues(int[] values) {
    values.filter(value -> value > 0);

    return;
}
```

10. Place the *values.filter(value -> value > 0)* directly into the return statement. Why? We can return the result of the filter directly to the return of the method. How? Placing the statement after the return.

```
public static int[] filterOnlyPositiveValues(int[] values) {
    return values.filter(value -> value > 0);
}
```

6.3. Bypassing a temporarily invalid program

The step-by-step hints provided to the students will inevitably direct the student to a temporarily invalid program. The Refactor Tutor evaluates the program and verifies for correct syntax (static analysis). We can bypass the evaluation (dynamic analysis) by letting the Refactor Tutor know the student is currently busy with the refactoring to a higher-order function. With the IDEAS Strategy type, we can explicitly turn off the evaluation of the program when the student is engaged in the refactoring. At the last step of the refactoring, we can enable it again since we expect the refactored end solution to provide a valid result.

As for correct syntax, if we look at the steps, we will always produce valid syntax. However, in some cases we may want to provide additional steps that lead to placeholder variables. For example, a lambda expression for a novice programmer may be hard to understand. With the placeholder variables, we can introduce smaller grained steps that may lead to, for example, *value -> ?* with ? being a placeholder variable that can be later filled in. AskElle, a functional programming based tutoring system built on the same framework (IDEAS) as the Refactor Tutor, also introduces 'holes', which are placeholder variables to learn students step-by-step functional programming Gerdes et al. [2016]. We will now refer to placeholder variables as holes.

6.3.1. Bypassing the evaluation

When the student engages in the refactoring to a higher-order function, we need to bypass the evaluation of the program until the student finishes the last step. We can do so by implementing the following:

- Introducing a minor Rule in our Strategy as the first rule. A minor Rule is a Rule that is executed without needing any intervention from the student. The minor rule must save a Ref type (reference) with a specific identifier (such as turnOffEval) into the Environment type. The Environment stores the environment variables which can be accessed during the exercises.
- Introducing another minor Rule which removes the Ref with the specific identifier turnOffEval from the Environment. The minor Rule must be included in our Strategy as the second last step since the evaluation would need to be turned on again after the completion of the last step.
- When trying to determine whether we need to evaluate the program (in the diagnoseR service), we need to include an additional check to turn the evaluation off if the Environment contains a Ref with the specific identifier turnOffEval.

6.3.2. Introduction of holes

An older version of the Refactor Tutor already made use of holes [Keuning et al., 2014]. However, the current version does not support holes anymore. For this refactoring, we have decided to re-introduce holes in the Refactor Tutor again. We had to make a few changes:

- Implemented the Term implementation of the Syntax Hole type
- Implemented the AST and its Term implementation of the Hole type

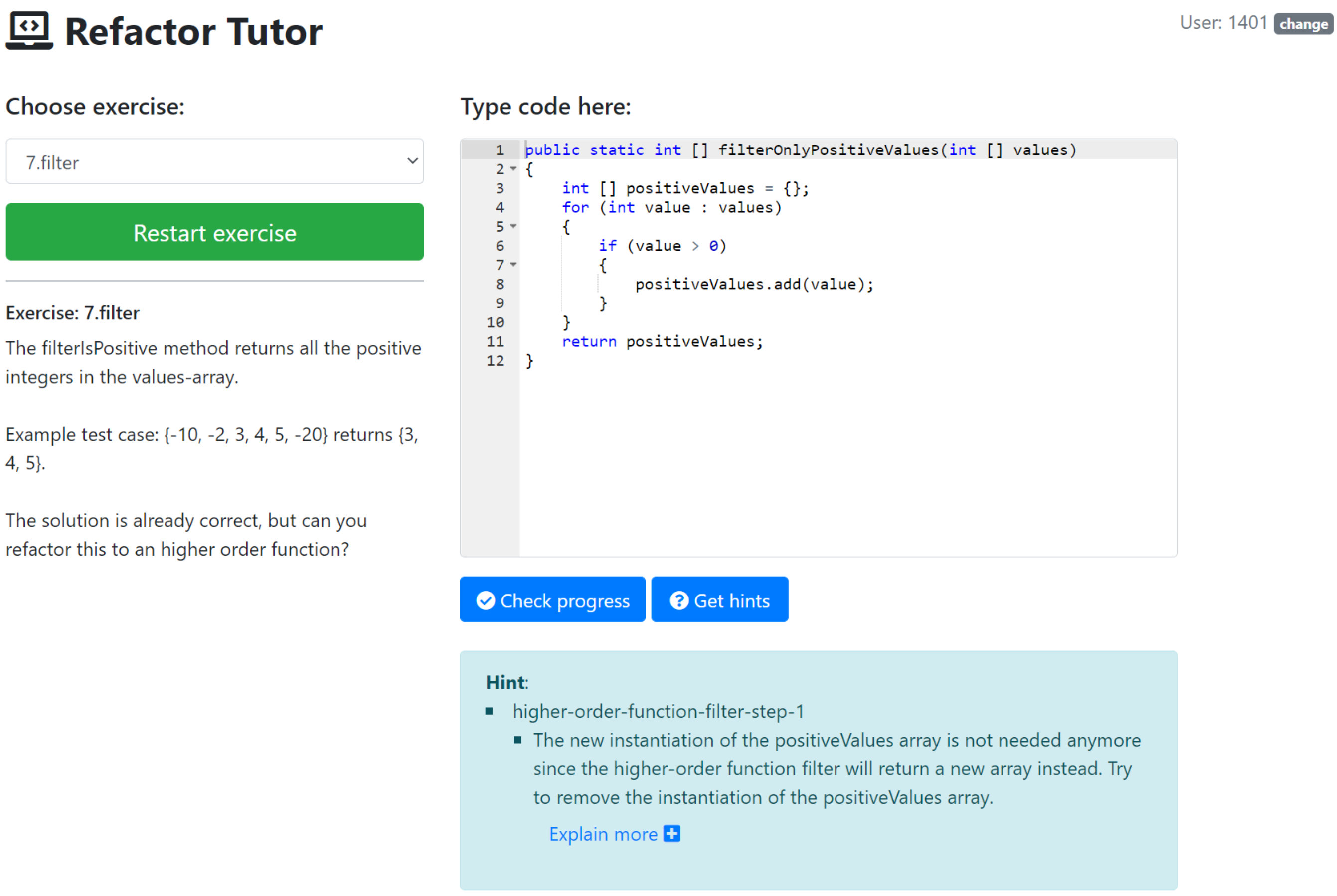
6.4. Steps in the Refactor Tutor

We have tried to implement the step-by-step hints for the *map* and *filter* exercises in the Refactor Tutor, but we were unable to do so. It required too many changes to the Refactor Tutor to get it working given our time constraint. The Refactor Tutor has been built with a specific goal in mind, that is to provide imperative code improvements that require a program transformation of some sort. For example, in the Refactor Tutor UI, the last underlying hint will always show the text *Try to use this example code* followed with the code that should be used (the code is directly derived from the program transformation). In our case, taking step 2 of the exercises as an example, we need to remove a piece of code. The last underlying hint of our step would need to show the text *Try to remove the following code* with the code that should be removed.

For future work, we have set up a list of items that need to be implemented in the Refactor Tutor to accommodate our step-by-step refactoring:

- Implement an information screen before starting the exercise. The information screen will be used for step 1, to introduce the concept of higher-order functions.
- Implement bypassing a temporarily invalid program. In section 6.3.1, we have written the steps down that need to be implemented to bypass the evaluation of the program.
- Extend the Refactor Tutor to handle different types of program transformations. As seen in the example above, the Refactor Tutor always expects the program transformation to be in one format. This extension requires hefty changes in the architecture of the Refactor Tutor.

- At last, the steps of the exercises need to be implemented.



Exercise: 7.filter

The filterIsPositive method returns all the positive integers in the values-array.

Example test case: {-10, -2, 3, 4, 5, -20} returns {3, 4, 5}.

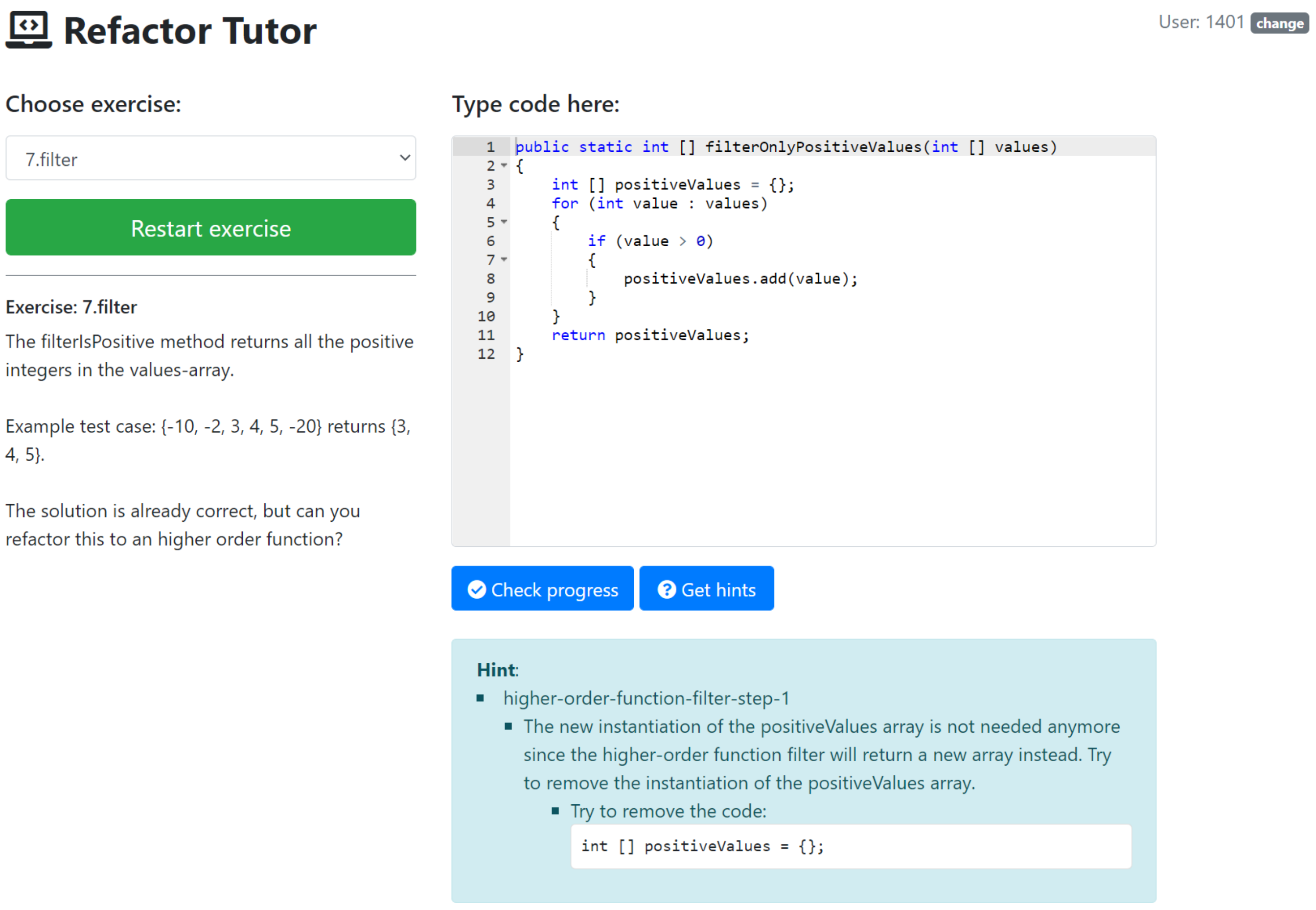
The solution is already correct, but can you refactor this to an higher order function?

```java
public static int [] filterOnlyPositiveValues(int [] values)
{
    int [] positiveValues = {};
    for (int value : values)
    {
        if (value > 0)
        {
            positiveValues.add(value);
        }
    }
    return positiveValues;
}
```

Hint:
- higher-order-function-filter-step-1
  - The new instantiation of the positiveValues array is not needed anymore since the higher-order function filter will return a new array instead. Try to remove the instantiation of the positiveValues array.

    Explain more

Figure 6.1: Refactor Tutor - Example of how step 2 first hint of the filter exercise would look like



Exercise: 7.filter

The filterIsPositive method returns all the positive integers in the values-array.

Example test case: {-10, -2, 3, 4, 5, -20} returns {3, 4, 5}.

The solution is already correct, but can you refactor this to an higher order function?

```
public static int [] filterOnlyPositiveValues(int [] values)
{
    int [] positiveValues = {};
    for (int value : values)
    {
        if (value > 0)
        {
            positiveValues.add(value);
        }
    }
    return positiveValues;
}
```

Hint:

- higher-order-function-filter-step-1
  - The new instantiation of the positiveValues array is not needed anymore since the higher-order function filter will return a new array instead. Try to remove the instantiation of the positiveValues array.
    - Try to remove the code:

```
int [] positiveValues = {};
```

Figure 6.2: Refactor Tutor - Example of how step 2 underlying hint of the filter exercise would look like

7
Automated layered feedback compared to teachers' feedback

This chapter relates to research question 4: How similar is the layered automated feedback to the feedback that teachers would provide?

In our previous chapter, we have set up the step-by-step hints to refactor imperative loops to higher-order functions *map* and *filter*. In this chapter, we evaluate our feedback to the feedback that teachers would provide.

7.1. Structure of our interview

The interview is semi-structured. We have a structure in place for the interview, but we try to keep it as interactive as possible and we may deviate from the questions asked depending on the answers of the teachers. The interview takes around 40 minutes.

The interview is separated into four parts: the introduction, general questions, automated steps, and final evaluation. Before the interview, the teachers are provided with a small introduction of the interview and some screenshots of the Refactor Tutor UI with the step-by-step hints of the refactoring.

7.1.1. Introduction

We start with a small introduction on the subject (2 min). The introduction contains the following:

- The bigger picture, teaching students with an imperative programming background functional programming concepts
- Teaching through the use of an Intelligent Tutoring System, the Refactor Tutor
- In particular, refactoring imperative code to the higher-order functions map and filter

7.1.2. General questions

The general questions would allow us to understand the background of the teacher and his opinions/experience on the particular subject (8 min).

- How would you teach students with an imperative programming background functional programming concepts? In particular, higher-order functions such as map and filter?
- What is your personal experience with Intelligent Tutoring Systems?
  - If it is positive, do you think it lessens the workload of teachers?
- Do you think we can improve teaching students with an imperative or functional programming background the other programming paradigm?
  - And if so, how?
  - And if not, do you think we can leverage the other programming paradigm to teach the other?
  - And do you think this would be possible with the use of an Intelligent Tutoring System?

7.1.3. AUTOMATED STEPS

We now go to the steps that the prototype would provide. For every step, we would like to know what the teachers think (15-20 min). Questions that may be asked are as follows:

- Is it clear?
- Can it be more clear?
- Do you think this step is easy to understand for students?
- Does it need to be smaller?
- Does it need to be in another order?
- Are the why and how of the step understandable?
- Would you also provide this step in this way?

7.1.4. FINAL EVALUATION

For the final evaluation (5-10 min) we try to get some more feedback on what the teachers think of the prototype with questions such as:

- What do you think of the prototype? (usefulness, usability, practicality)
- Have some of your negative opinions (from the general questions) changed after seeing some screenshots of the prototype?
- Is something unclear?
- What can be improved?
- Is something missing?
- Do you see yourself using it in your courses?
- When do you think this should be applied? (course level)

7.2. Interview

We have interviewed 2 teachers. The interviews are recorded and are transcribed. The transcripts can be requested. We will be referring to them as teachers X and Y.

7.3. Results

7.3.1. Teacher X

Background

X is an assistant professor at the Open University of the Netherlands. X does not have an academic background in refactoring and functional programming, but has done some research in easing/helping students to become a better programmer. X comes from an imperative programming background with some experience in functional programming. X's opinion on Intelligent Tutoring Systems is that they are great for automating some aspects in the learning process, but that these systems miss the personal aspect and treat all students equally.

When trying to teach imperative to functional programming, X would also drill down on the why aspect. Furthermore, X believes that teaching functional programming to students with an imperative programming background can be improved.

Feedback on the steps

X had two feedback points on the steps. First, X would have removed all the imperative code step-by-step before anything else. This relates to step 9, where we remove the if statement only after the if condition has been moved to the filter method. X argues that the student already knows what the program does, so step 9 can be placed after step 4.

Secondly, X argues that the cognitive load of learning lambda expressions is too much for the student to handle. We are already introducing a new concept of higher-order functions so that should be the focus instead. The students should not learn lambda expressions here. X advised using anonymous methods instead. After the students have learned lambda expressions, the students can apply their knowledge to implement the exercises with lambda expressions to show the real-world applicability of lambda expressions.

Feedback on prototype

X could see the prototype being adapted into certain courses since students will benefit from this. X mentioned that we could use it in the beginner Java course or the Python pre-master course at the Open University of the Netherlands. The latter would be more applicable since it is under development and higher-order functions are also used there.

X asked whether complex exercises would work and what kind of adjustments would be needed to accommodate complex exercises. The answer to this question was that the prototype currently supports simple cases, but could very well be extended to support complex exercises.

Finally, X tried to make it clear that the students should really understand the feedback and recommended looking at the most frequent student misconceptions when students transform imperative code to functional. These misconceptions can then be used to formulate and adjust the feedback.

7.3.2. Teacher Y

Background

Y is an assistant professor at the Open University of the Netherlands. One of Y's fields of research is refactoring. Y has tried on multiple occasions to introduce functional programming concepts into courses, the latest being a Javascript web application development course on the Open University of the Netherlands. In this course, imperative programming is introduced first, and then a functional style. The course tries to push students to a functional style by providing the advantages of a functional style (Y mentioned smaller code, better readable, better maintainable, and better extendable) and the disadvantages of not using a functional style. Furthermore, in the practical exercise of the course, there is a mandatory criterion that there should not be any loops in the program.

Y has been struggling for a long time to improve the transition of going from imperative programming to functional programming and argues that within education there needs to be more done to highlight the multiple programming paradigms, but that there is unfortunately limited time to teach students all these subjects.

As for Intelligent Tutoring Systems, Y has never worked with them before.

Feedback on the steps

Y found it very hard to provide feedback on the steps since he does not know if it is logical and understandable for students. Y also mentioned that this is a very complex subject since functional programming requires a completely different thinking model compared to imperative programming. Y needed much more time to think about the steps and to work out cases. Nonetheless, Y did had some remarks on the steps.

The first remark was that the higher-order function filter should be explained more explicitly. What you do with each element must be clear and that it is a boolean function (predicate). Secondly, the problem with step 2 is that the program does not run anymore. Y questions whether this is what you want and if it is possible to refactor where each step still produces a program that can be run but also does not know the answer to this question. Finally, Y argues that we are introducing too much complexity with lambda expressions and that we are trying to do two things at the same time. Lambda expressions and predicates should be taught separately and before this exercise. When students understand these topics, they can apply them here. Y would not make use of anonymous methods since lambda expressions are eventually what should be used.

Feedback on prototype

Y thinks that the prototype is very useable and applicable if you want students to learn to refactor. He mentions that there is a lot of imperative code that can be refactored to functional code. Refactoring has been a subject of many courses, but not the main subject. Should there be a course where refactoring is the main subject, then Y can see the prototype being used. However, as Y mentioned before, there are just too many subjects to handle.

7.4. Conclusion

From both interviews, we can conclude a few things. Both teachers think that teaching the transition of imperative programming to functional programming can be improved. Y argues that there are just too many subjects to handle and that if refactoring becomes the primary subject of a course, then the prototype could be used. X mentioned and proposed that we could use the prototype in existing courses, such as the pre-master Python course since they also use higher-order functions there.

Both teachers had the same remarks on the introduction of lambda expressions. In this exercise, we should not introduce lambda expressions. They should be introduced beforehand so that they can apply it in this exercise. Furthermore, they both questioned whether these steps are logical and understandable for students since it is complex (switching from one programming paradigm to another one). This makes it clear that we should interview students as they are our primary end-users.

8

Conclusion

The main question of our research is as follows: *How can we provide layered automated feedback to support students to learn to refactor imperative loops to higher-order functions?*

With the sub-questions:

RQ1 *What can be found in the literature about teaching students with an imperative programming background functional programming concepts with the possible use of automated systems?*

RQ2 *How do we implement the refactorings of imperative loops to higher-order functions in the Refactor Tutor?*

RQ3 *How do we transform these refactorings to layered automated feedback?*

RQ4 *How similar is the layered automated feedback to the feedback that teachers would provide?*

8.1. RQ1: Current situation

From research question 1 we have gathered that there is a lot of ongoing research on learning students functional programming concepts. We have found the domain of multiparadigm programming that combines multiple programming paradigms. The work of Ross [1998] tries to teach functional programming concepts by applying them at an early stage to imperative programming. Furthermore, we see an emergence of multiple programming paradigms being combined into a single programming language. However, we have not found a multiparadigm programming tutoring system that focuses on teaching functional programming concepts by leveraging the imperative programming background of students. We believe that this further highlights the importance of our work.

8.2. RQ2: Implementation of the refactorings

In research question 2 we extended the Refactor Tutor to support our refactoring. We have done the following:

- Built-in support for lambda expressions in the internal data types.

- Built-in support for (higher-order) functions.
- Created pre-conditions and rules of when the refactoring should take place.
- Added the program transformation of our refactoring.
- Set up two example exercises for the higher-order functions *map* and *filter*.
- Adapted our program transformation into the hint system of the Refactor Tutor.
- Resolved last issues with showcasing our refactoring on the Refactor Tutor UI.

8.3. RQ3: Layered automated feedback

In research question 3 we try to provide step-by-step hints for our refactoring so that it is clear for the student why and how our refactoring takes place. We set up steps for the exercises to refactor to the higher-order functions *map* and *filter*. The steps are defined based on the general feedback guidelines of the Refactor Tutor. The steps will inevitably direct the student to a temporarily invalid program. Because of this, we need to turn off the evaluation of the program when the student starts with our refactoring and turn it on again to compare whether the output of the refactored solution is the same as before.

Furthermore, we have introduced placeholder variables known as holes for smaller grained steps. An older version of the Refactor Tutor already made use of holes, but the current version does not support it anymore. We have therefore re-introduced holes in the Refactor Tutor again.

Unfortunately, given our time constraint, we were unable to implement the step-by-step hints in the Refactor Tutor as it required some hefty changes. However, we have laid out the steps that need to be implemented for accommodating our step-by-step hints.

8.4. RQ4: Similarity to the feedback of teachers

In research question 4 we compare the step-by-step feedback of our refactoring against the feedback that teachers would provide. We interviewed two assistant professors of the Open University of the Netherlands on whether they would provide the same step-by-step feedback and what their opinion is of the prototype. Both teachers were positive of the prototype and see it being used in certain courses. They also had some feedback on the steps, but the most notable feedback was that they both thought that introducing lambda expressions would cause for too much complexity and that lambda expressions should be introduced elsewhere, preferably before the exercise. Furthermore, they both questioned whether these steps are logical and understandable for students since it is complex (switching from one programming paradigm to another one). This makes it clear that we should interview students as they are our primary end-users.

8.5. Conclusion

With the Refactor Tutor, an Intelligent Tutoring System, we teach students with an imperative programming background functional programming concepts, in particular higher-order functions. There has been research on teaching students multiparadigm programming concepts, but not with an Intelligent Tutoring System. We believe that this further highlights the importance of our work.

We extended the Refactor Tutor to support our refactoring and set up step-by-step hints for the student to understand our refactoring. Unfortunately, we were unable to implement the step-by-step hints in the Refactor Tutor given our time constraint. We then evaluated the step-by-step hints of the system with the step-by-step hints the teachers would provide. There were some things that they would do differently, the most notable feedback was that lambda expressions should not be introduced in the exercise, but beforehand. Furthermore, the teachers questioned whether the hints would be logical and understandable for the students since we are going from one programming paradigm to another, which requires a different thinking mindset. This is further discussed in the next and final Chapter: Discussion.

9 Discussion

Our research contributes to a new way of teaching multiparadigm programming, specifically from imperative programming to functional programming, with the use of an Intelligent Tutoring System. There has been ongoing research on teaching students multi-paradigm programming, but none so far using Intelligent Tutoring Systems. However, it may be that there is research not published yet. Our research is also in line with the goal of the Refactor Tutor, which is improving code quality. We encountered some issues with the Refactor Tutor UI that had to be resolved to showcase the refactoring. We had to add the following:

- The Syntax ArrayDecls and ArrayDecl types Term' implementations were added.
- Support for the Syntax Lambda type was added to the Printer.

The step-by-step hints were set up based on the general guidelines of the Refactor Tutor. However, it may be that these general guidelines are not appropriate for our refactoring. These guidelines have been set up based on refactoring imperative code. Our refactoring goes from one programming paradigm to another, which requires a different thinking mindset. The didactic side of our refactoring can be further researched, which would cost additional time. Our approach is an agile method where we validate the prototype with its users at its earliest.

The prototype of the step-by-step hints still needs to be implemented, but we have laid out the steps that need to be taken to support it. Furthermore, our prototype also needs some additional work to be used in real-world scenarios. This includes:

- Extending our prototype to support more complex cases to eventually create more complex exercises.
- Integrating our work with the work of the Refactor Tutor, specifically the hint system.
- Handling edge cases of the step-by-step hints.
- Improving the code quality of the prototype.

Two teachers have been interviewed from the Open University of the Netherlands. Based on the sample size, the margin of error increases, which should be taken into consideration.

In our exercises, the concept of lambda expressions is introduced. However, the teachers that were interviewed argued that this will bring in too much complexity for the student to handle. The student should already know of lambda expressions so that they can apply them in our exercise. With this feedback, the steps that are related to introducing the higher-order function can be removed. For the map exercise, step 1 could be removed and step 6 and 7 can be merged together since we do not need to carefully introduce higher-order functions. Furthermore, the teachers questioned whether the step-by-step hints are logical and understandable enough for the students to understand the refactoring. To validate this, we need to interview students and let them try out the prototype. With the results, we can determine whether it is logical and understandable enough and adjust our automated feedback based on students' feedback. As the teachers mentioned, the prototype could be used in one of their courses (note that the student must first understand higher-order before using the prototype).

Finally, it would be interesting to research the most frequent student misconceptions when students transform imperative code to functional. These misconceptions can then be used to formulate exercises and adjust feedback.

Bibliography

Paolo Antonucci, Christian Estler, Durica Nikolić, Marco Piccioni, and Bertrand Meyer. An incremental hint system for automated programming assignments. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '15, page 320–325, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334402. doi: 10.1145/2729094.2742607. URL https://doi.org/10.1145/2729094.2742607. 8

Morgan C. Benton and Nicole M. Radziwill. Improving testability and reuse by transitioning to functional programming. *CoRR*, abs/1606.06704, 2016. URL http://arxiv.org/abs/1606.06704. 1, 3, 4

T. Budd. *Multiparadigm programming in Leda*. Addison-Wesley, 1994. 15

Ike Casteleyn. Improving maintainability in JavaScript. Master's thesis, Open Universiteit, November 2019. 5, 7, 12, 18, 21

Manuel Chakravarty and Gabriele Keller. The risks and benefits of teaching purely functional programming in first year. *J. Funct. Program.*, 14:113–123, 01 2004. doi: 10.1017/S0956796803004805. 14

Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L. Thomas van Binsbergen. Ask-Elle: an adaptable programming tutor for Haskell giving automated feedback. *International Journal of Artificial Intelligence in Education*, 27, 02 2016. doi: 10.1007/s40593-015-0080-x. 14, 36

Alex Gyori, Lyle Franklin, Danny Dig, and Jan Lahoda. Crossing the gap from imperative to functional programming through refactoring. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 543–553, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450322379. doi: 10.1145/2491411.2491461. URL https://doi.org/10.1145/2491411.2491461. 6, 12, 20

I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, pages 30–39, 2007. doi: 10.1109/QUATIC.2007.8. 5

J. Jansen, Ana Oprescu, and M. Bruntink. The impact of automated code quality feedback in programming education. *Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE 2017*, 2018. 2, 3, 5, 6

S. Joosten, K. Berg, and G. Hoeven. Teaching functional programming to first-year students. *J. Funct. Program.*, 3:49–65, 1993. 14

Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. Strategy-based feedback in a programming tutor. In *Proceedings of the Computer Science Education Research Conference*, CSERC '14, page 43–54, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450333474. doi: 10.1145/2691352.2691356. URL https://doi.org/10.1145/2691352.2691356. 37

Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. Code quality issues in student programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '17, page 110–115, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450347044. doi: 10.1145/3059009.3059061. URL https://doi.org/10.1145/3059009.3059061. 2, 5, 6

Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education*, 19:1–43, 09 2018. doi: 10.1145/3231711. 2, 5, 15

Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. How teachers would help students to improve their code. pages 119–125, 07 2019. ISBN 978-1-4503-6895-7. doi: 10.1145/3304221.3319780. 3, 7, 8, 12, 13, 30

Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. Student refactoring behaviour in a programming tutor. In *Koli Calling '20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research*, Koli Calling '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450389211. doi: 10.1145/3428029.3428043. URL https://doi.org/10.1145/3428029.3428043. 6

Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. *A Tutoring System to Learn Code Refactoring*, page 562–568. Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450380621. URL https://doi.org/10.1145/3408877.3432526. 1, 2, 6, 7, 8, 9, 10, 12

J.T. Kristensen, M.R. Hansen, and H. Rischel. Teaching object-oriented programming on top of functional programming. In *31st Annual Frontiers in Education Conference. Impact on Engineering and Science Education. Conference Proceedings (Cat. No.01CH37193)*, volume 1, pages TID–15, 2001. doi: 10.1109/FIE.2001.963848. 14

Claudio Mirolo and Cruz Izu. An exploration of novice programmers' comprehension of conditionals in imperative and functional programming. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '19, page 436–442, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368957. doi: 10.1145/3304221.3319746. URL https://doi.org/10.1145/3304221.3319746. 3

Susanne Narciss. The impact of informative tutoring feedback and self-efficacy on motivation and achievement in concept learning. *Experimental psychology*, 51:214–28, 02 2004. doi: 10.1027/1618-3169.51.3.214. 2

Susanne Narciss. *Feedback Strategies for Interactive Learning Tasks*, pages 125–144. 01 2008. 8

Martin Odersky et al. The Scala programming language. *URL http://www. scala-lang. org*, 2008. 15

Francisco Ortin, Jose Manuel Redondo, and Jose Quiroga. Design and evaluation of an alternative programming paradigms course. *Telematics and Informatics*, 34(6):813–823, 2017. ISSN 0736-5853. doi: https://doi.org/10.1016/j.tele.2016.09.014. URL https://www.sciencedirect.com/science/article/pii/S0736585316301393. SI: IT Education Training. 15

D. F. Ross. Using functional paradigms in an imperative language. Department of Computer Science, University of Karlstad, 1998. 15, 45

Valerie J. Shute. Focus on formative feedback. *Review of Educational Research*, 78(1): 153–189, 2008. doi: 10.3102/0034654307313795. URL https://doi.org/10.3102/0034654307313795. 8

Alexandria Katarina Vail and Kristy Elizabeth Boyer. Identifying effective moves in tutoring: On the refinement of dialogue act annotation schemes. In Stefan Trausan-Matu, Kristy Elizabeth Boyer, Martha Crosby, and Kitty Panourgia, editors, *Intelligent Tutoring Systems*, pages 199–209, Cham, 2014. Springer International Publishing. ISBN 978-3-319-07221-0. 8

Kurt VanLehn. The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems. *Educational Psychologist*, 46:197–221, 10 2011. doi: 10.1080/00461520.2011.611369. 2