AUTOMATIC FEEDBACK ON COMMON LOGIC ERRORS

HOW TO USE UNIT TESTS TO PROVIDE FEEDBACK TO STUDENTS WHEN REFACTORING

Toonen, M. (Meine)

Award date:
2021

AUTOMATIC FEEDBACK ON COMMON LOGIC ERRORS

HOW TO USE UNIT TESTS TO PROVIDE FEEDBACK TO STUDENTS WHEN REFACTORING

by

Meine Toonen

To be defended publicly on Friday 29 January, 2021 at 10:30 AM.

Name:	Meine Toonen
Student number:	
Course code:	IM9906

Open Universiteit
www.ou.nl

AUTOMATIC FEEDBACK ON COMMON LOGIC ERRORS

HOW TO USE UNIT TESTS TO PROVIDE FEEDBACK TO STUDENTS WHEN REFACTORING

by

Meine Toonen

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

Open University of the Netherlands,
Faculty of Science
Master's Programme in Software Engineering

Student number:	852116309	
Course code:	IM9906	
Graduation committee	dhr. prof. dr. Johan Jeuring (chairman)	Open University
	dhr. dr. Bastiaan Heeren (first supervisor)	Open University
	mw. dr. Hieke Keuning (second supervisor)	Utrecht University

Open Universiteit
www.ou.nl

ACKNOWLEDGEMENTS

During my first study at the Hague University, I missed something: a more theoretical approach to my learning. It was fun and I learned a lot, but I knew: someday I will wanted to get my masters degree. And when the opportunity opened up to do it, I took it and went towards my goal.

I would first like to thank my committee. Bastiaan, thanks for all of your (detailed!) input, positive feedback and pleasant coaching skills. It was a joy creating my thesis under you tutelage. Hieke, even though you were busy finishing your PhD and settling in your new job, you found the time to provide feedback, and even helped labelling! Much appreciated. And Johan, thank you for your feedback and suggestions into the broader aspects of my research. You all helped driving me to create a better research, thank you for that.

And lastly Sara. Where would I be without you? Probably still Gouda. I loved sparring about research, Machine Learning, trees and forests. Thank you for helping and challenging me, and allowing me to be a better student. I love you and Apekop.

This was a fun journey.

Meine Toonen
Culemborg, 2021

- Arriving somewhere, but not here -

SW

CONTENTS

ABSTRACT

Because of time constraints, higher education teachers spend little time teaching about code quality. To bridge this gap, teachers can use Intelligent Tutoring Systems: applications designed to help training some skill by providing feedback on exercises. This research studies how unit tests can help identify common logic errors students make when they practise refactoring in a tutor, and how unit tests can help to provide feedback on those errors.

This research continues to study an existing tutor: the refactor tutor. Using a literature review as a starting point, we identified common logic errors made in a previous study on the refactor tutor. We labelled the errors, and validated them by letting multiple experts check those labels. We developed unit tests for exercises from that study, and trained binary decision trees to identify six common logic errors. Using a questionnaire, we consulted with teachers to verify that the feedback on the recognised common logic error, is on par with what the teachers would give. Using various verification methods, we determined that at least two common logic errors can be recognised, and that we can provide valuable feedback on those errors.

We showed that unit tests are a viable option to recognise common logic errors, and therefore tutors can use these tests for providing feedback to students who are refactoring. If a tutor should implement this, teachers would spend less time correcting common mistakes and providing feedback to students, but students are still able to learn from their mistakes.

1 INTRODUCTION

Creating software is a diverse craft, ranging from design, implementation and maintenance – and everything in between. According to Bruce [2018], the number of students in Computer Science and Software Engineering is rapidly rising and teachers have trouble keeping up with these numbers. The consequence of this, is that some aspects of software development are neglected: in a study done by Kirk et al. [2020], it is shown that code quality is one of those aspects that receive less attention. According to Boehm et al. [1976], bad code quality leads to difficult to maintain software, resulting in steep increases of maintenance costs. Luckily, teachers can employ Intelligent Tutoring Systems: software specifically designed to teach some skill, without heavy reliance on the supervision of teachers.

In this research we study a tutor aimed at teaching refactoring: the Refactor Tutor. This is a simple web application, developed by Keuning et al. [2020]. This tutor presents the learner – students Computer Science or Software Engineering, after one or two courses of programming – with a total of six exercises. Each exercise is a functionally correct method, which the student has to improve the quality of. At each step the student can ask for feedback, and the progress is checked. The tutor provides guidance on the refactoring steps the students have to make, by giving feedback automatically, and providing a way students can actively check their progress. It has some rudimentary unit tests to check for the correctness of the submission, but provides little feedback when an error is made.

Throughout this thesis we use the term “common logic error”. There are two parts to this term: the first – common – describes those errors that often occur in a previous study. This ensures that we have enough data to make determinations about the errors. The second part – logic – concerns the type of error. Logic here means that it can be observed by unit tests, as opposed to style errors: errors dealing with indents, capitalization and variable names.

The second definition we use is about the term refactoring. We adhere to the definition given by Fowler [2018]: “Refactoring is the process of changing a software system in a way that does not alter the external behaviour of the code yet improves its internal structure”.

This study is aimed at providing students feedback on commonly made mistakes when they practise refactoring. Our hypothesis is that students often make the same mistakes when refactoring, and that we can observe these mistakes by employing unit tests. When we know recognize a common error, we can provide feedback to the student, and the student can make an improvement and learn from its mistake. When feedback on common errors can be given automatically, teachers have to spent less time educating on those errors. This reduces the workload, and frees up time for other areas in the curriculum.

By performing a literature study, we collected common errors students make when programming in general. Using these errors as a hierarchy, we created a list of errors students make when they practise refactoring. We labelled those errors, and we verified those labels, by letting other experts perform the same labelling task on a subset. We wrote various unit tests for the exercises and used the obtained labels to train a binary decision tree. This decision tree allows us to recognise a common logic error. We verified the outcome of the models by checking them against a test set. Lastly, we sent out a questionnaire to teachers to check if the feedback we created on the errors are applicable.

Our contribution is that we have created a list of common logic errors students make when refactoring, developed an algorithm to recognise common logic errors, and showed that the feedback we can automatically provide is sound.

The remainder of this document is as follows: Section 2 provides context on tutors, feedback and methods we employed in our research. Section 3 presents our research questions and outlines the methodology used, and Section 4 presents the results obtained. In Section 5 we discuss the results, and give our thoughts on further research, and in Section 6 we present our conclusion.

2 CONTEXT

2.1. LITERATURE

In this section we look at the current literature concerning feedback, tutors and how learning works.

2.1.1. SOFTWARE QUALITY AND REFACTORING

Software is becoming more and more important in our society. For example, our transportation, electricity and work depend on it. It is therefore important that the software we use is of good enough quality, because according to Boehm et al. [1976], good quality software can save money in the overall lifespan of the software. These quality aspects can be divided into multiple categories and subcategories. These aspects are described by Boehm et al. [1976] as a characteristics tree. For example, the main category maintainability has the subcategories testability, understandability and modifiability.

To make the quality of a piece of software measurable, there are metrics. For understandability, one can use complexity as a measure: if some function is very complex, it will be harder to understand. This can be quantified by the cyclomatic complexity metric, as defined by McCabe [1976].

When some function is becoming, for example, too large or too complex, a programmer can decide to refactor said function. Refactoring is changing code, without changing its behaviour, with the purpose of making the code more maintainable [Fowler, 2018]. To guarantee the stability of the behaviour, Mens and Tourwé [2004] state the pragmatic solution of rigorous testing. When the tests succeed after the refactoring, there is good evidence the behaviour has not changed.

2.1.2. SEEKING HELP AND RECEIVING FEEDBACK

Students typically learn from their teachers by attending lectures and reading course material. In addition to this, exercises can be made, on which the teachers provide some kind of feedback to the learners. This feedback can fall into two categories: summative and formative. Summative feedback provides some sort of evaluation on how well the learner mastered the material, for example a grade.[1] Formative feedback on the other hand, is

[1] `https://sites.tufts.edu/teaching/assessment/assessment-approaches/formative-and-summative-feedback`

typically not about grades and more about changing the thinking of the learner.

Shute [2008] did an extensive literature review to gain more understanding about the workings of formative feedback and on how it could be best applied. She analysed 138 articles about formative feedback, and made multiple suggestions on what to do and what *not* to do when giving feedback, when to give feedback and how to approach giving feedback depending on some characteristics of the learner (i.e. is it a high or a low achieving learner).

When students learn something new, or are working on a problem, they can discover that they do not have sufficient knowledge. This process is called meta cognition: knowing what you do not know and (how) to act on it. Aleven [2013] describes a multitude of ways students can seek help: ask the teacher, search the internet, look at Wikipedia, etc. For help to be effective, it must explain the why of a problem. VanLehn [2006] showed that if the feedback only shows correct/incorrect, the student will give up without learning anything. Therefore, the feedback should be tailored to the student: it must match the level of mastery of the student on the subject. This is called scaffolding. The implication of this is that there must be multiple degrees of feedback: from coarse-grained to fine-grained. For the learning event to be effective, the teacher should gradually remove the scaffold when the learner is becoming more proficient. As Harris et al. [2009] conclude, this adaptive behaviour in providing feedback has a positive effect on the mastery of a subject.

2.1.3. Intelligent Tutoring Systems

Here we look at Intelligent Tutoring Systems: how they are designed and used, and which types exist.

Design of Intelligent Tutoring Systems

When teaching, a tutor system can be a large asset. It has the capability to assess the progress and knowledge of students, without the need for a teacher to intervene. VanLehn [2006] studied multiple tutoring systems, differing in domain: physics, algebra, SQL and practical training about the operation of systems. VanLehn describes some common traits shared by intelligent tutor systems. The main components as found in ITSs are:

The outer loop

The **outer loop** is the part of the tutor which selects the task to be completed next. A task is an exercise, usually taking between a couple of minutes and an hour to finish. VanLehn describes four levels of complexity for selecting the next task: at its simplest, it is a list with tasks to be executed in sequence, and at the most complex end of this scale is a system based on *macroadaption*: the system picks the next task based on the missing knowledge of the student. For macroadaption to work, the ITS must keep track of the progress of the student: this is called the student model.

The inner loop

The **inner loop** is concerned with the steps inside a task. A step is a user interface action the user takes, and is part of solving the task. The steps available on most tutors are: providing minimal (correct or incorrect) and error specific (helping the student understand what is wrong) feedback, next step hint (what to do next), assessment of knowledge (what does the student know) and review of the whole solution.

Feedback and hints

For **feedback and hints**, the difficulty lies in *when* to give them and the granularity of the feedback/hint. As for the when, there are three types: immediate, delayed and on request. Ideally, hints and feedback are given only when the student needs it. For the granularity – ranging from very coarse (stating the correctness) to very fine (it gives the answer) – the level of proficiency of the student must be leading.

CURRENT STATE OF PROGRAMMING TUTORS IN EDUCATION

In 2018, Keuning et al. [2018] did a systematic literature review on several aspects of 101 tools providing feedback on student submissions of programming exercises. They set out to determine what type of feedback was given to the students using the tool, which techniques were used to create that feedback, and how teachers could expand on the exercises and feedback in the tool. This was done by reviewing 146 papers, regarding 101 tools, and labelling the different aspects of each tutor.

The article shows that the type of feedback most prevalent in such a tool is about mistakes, and of that, it was mostly about test failures (program is incorrect) and feedback on errors (program does not show behaviour the exercise expected) in the solution. When looking at the types of feedback of the previous decades, we see a declining trend for solution errors and a small rise in test failures.

A technique used to generate feedback is automated testing, where a solution from the student is checked against a predefined set of tests. Mitrovic et al. [2003] note that one or more unit tests can be used to observe a buggy rule. A buggy rule is a step in the (refactoring) process that is incorrect, but common enough to describe. Unit tests can thus be used to identify common errors.

Two other often used techniques are program transformation and basic static analysis. When using program transformation, the solution can be rewritten to some normal form. This normal form can then be checked for errors. Basic static analysis can be done using metrics. The metric can be used to reason about the solution, to show what is wrong with it.

To be useful for teachers, it must be easy to add new exercises and feedback. Most of the systems considered use model solutions. In a dynamic analysis, model solutions are correct implementations of the exercise used to automatically generate the desired output. When used with static analysis, the structure of the correct model is used to check against that of the student. Another form of adding new content is by using test data. This can be in the form of scripts, or as unit tests.

One example of a tutor system that uses unit tests, is the online tool described in the article by Fischer and von Gudenberg [2006]. They developed an online assessment tool, to which students can upload their solutions. The solution is then checked for syntax, if it adheres to the specification and a number of functional tests are run. These functional tests are executed using the JUnit framework.[2] The tests can be defined by the teacher as mandatory, optional or even secret, and are run immediately.

REFACTORING ITS

Keuning et al. [2021] developed an Intelligent Tutoring System focused on refactoring in the imperative language Java. The goal for this ITS is to aid teachers in giving feedback to students. The tool is based on the IDEAS framework[3] and was designed to provide feedback

[2]`https://junit.org/`
[3]`http://hackage.haskell.org/package/ideas`

as efficiently as possible. How and what kind of feedback was given was determined by comparing professional tools, advice from teachers and current literature on this subject.

The ITS uses rules and strategies for implementing the feedback system, where a rule is a single step in the refactoring process and a strategy is a series of rules detailing a refactoring solution. These rules can be ordered in sequences, so that priorities can be given. This resulted in an ITS that addresses more points than professional tools – such as PMD and SonarQube – do, and takes the advice of teachers into account on how to provide feedback.

On the user interface part, the ITS allows students to enter their solutions and press a button to validate their solution. When a student needs more help, they can get hints in varying degrees of granularity: they are first presented with the most coarse-gained hint, and each successive hint is more fine-grained. With each press of a button, the current state is sent to the back-end and saved into a database for analysis.

Keuning et al. [2020] studied how students refactor and how they value code quality. The students tried to solve six exercises, which consisted of small code snippets that were functionally correct, but needed some improvements with regards to the quality of the code. The study was done with 133 students, all of them having taken programming courses.

The research shows that the students requested hints on various levels of coarseness and they had the most difficulty with complex control flows.

2.1.4. Log analysis

When presented with a log database from a system, the amount of data can be overwhelming. To handle that problem, some structured way is needed to draw conclusions about the data. Jansen [2006] described a method for extracting and analysing data from transaction logs, called Transaction Log Analysis (TLA). The formulated method is focused on transaction logs from search engines, Intranet and web sites. These logs can describe a multitude of actions, and can typically be used to investigate performance issues, information structure or how users interact with the system. TLA involves three stages: *collection*, *preparation* and *analysis*.

The *collection* stage is done at first, and aims at acquiring a dataset as unobtrusively as possible. This ensures that the behaviour of the subjects does not change, which would alter the outcome. *Preparation* is the preprocessing of the data: for example loading it into a relational database, clean up of the data, parsing of data structures and normalising (searching) episodes. The normalisation action is done to group individual entries in the database to one person/actor. The last step, *analysis*, is concerned with getting conclusions from the data as acquired and prepared in the previous stages.

2.1.5. Testing

One of the ways to verify if a function (or method) is correct, is by writing unit tests. This is a simple program that provides a known input, executes the function-under-test and compares the result to some expected value. Within the Java language, one of the libraries used for this is JUnit. Unit testing is at the heart of Test Driving Development (TDD), where you first write a test and then write the code. It is linked to an improved code quality, as the study by George and Williams [2003] showed. However, as Aniche and Gerosa [2010] observed that there are common mistakes when applying TDD, leading to a decrease in code quality.

Claessen and Hughes [2011] have developed a tool for the language Haskell, named QuickCheck. It is a small tool that implements a language construct that allows developers to explicitly define the relationship between the input and output of a function. This is called Property Based Testing. When the relationship is defined, the function-under-test will be executed with multiple, randomly generated inputs, and the resulting outputs are checked for their compliance to the defined relationship. An implementation for the imperative programming language Java exists, called jqwik [4].

2.1.6. Classifiers

Whereas unit tests may be applied to identify logic errors, there may not necessarily be a single unit test that can detect all occurrences of a certain error. More likely, a combination of several unit tests is applicable. Machine learning offers tools to automatically train classifiers: models that can be used to combine features (the results of various unit test), into a single classification.

One of those models is a binary decision tree. Murthy [1998] states that a binary decision tree can be used to represent links in the data, and use those links to classify new data.

In every node of the tree, a decision is made based on an input feature: here, whether a certain unit test has fired or not. Every data point is thus represented as a path through the tree, leading to a leaf where the final classification occurs.

This model is a proper fit to the classification problem at hand, because the features are binary (unit test success of failure). An additional benefit of using binary decision trees, is they can be graphically represented – via a tree structure – in an easy to understand, and useful manner.

2.2. Current state of the refactor tutor

In 2020, Keuning et al. [2020] created an ITS focused on refactoring called the refactor tutor. It has the capability to provide feedback on multiple levels of granularity. In the current state of the ITS, feedback is given on the steps taken to refactor the given method, and a basic unit test is executed to validate the output. The refactor tutor is a web application, from which a screenshot is shown in Figure 2.1. It currently has six exercises, all of them are described here. It must be noted that exercise *6.havethree* is somewhat different: the first five already have a functionally correct method to begin with, whereas exercise six has no correct implementation to start with. All of the given initial programs are listed in Appendix B. The exercise descriptions are:

[4]`https://jqwik.net/`

1.even: `countEvent(int[] values)`

The countEven method returns the number of even integers in the values-array.

Example test case: 1,2,3,4,5 returns 2. You don't have to deal with negative numbers.

2.sumvalues : `sumValues(int [] values, boolean positivesOnly)`

The sumValues method adds up all numbers from the values-array, or only the positive numbers if the positivesOnly boolean parameter is set to true.

Example test case: calling sumvalues with 1,2,3,4,-5 and true returns 10.

3.oddsum: `oddSum(int [] array)`

The method oddSum returns the sum of all numbers at an odd index in the array parameter, until the number -1 is seen at an odd index.

Example test case: 44, 12, 20, 1, -1, 3, 5,-1, 99, 4 returns 16 (12+1+3)

4.score: `calculateScore(int changes, int day)`

The calculateScore method calculates the score for a train trip. The highest score is 10. The score is based on the number of changes and the day of the week (Monday is 1, Sunday is 7).

Dutch Railways (NS) has designed the following calculation: Base score: 10 For each change: -1 Trip on a weekday: -3

Example test case: for a trip with 2 changes on a Wednesday (day 3), calculateScore(2, 3) returns a score of 5 (10-2-3)

5.double: `hasDoubled(double savings, int interest)`

Write a program that calculates in how many years your savings have doubled with the given interest (as a percentage).

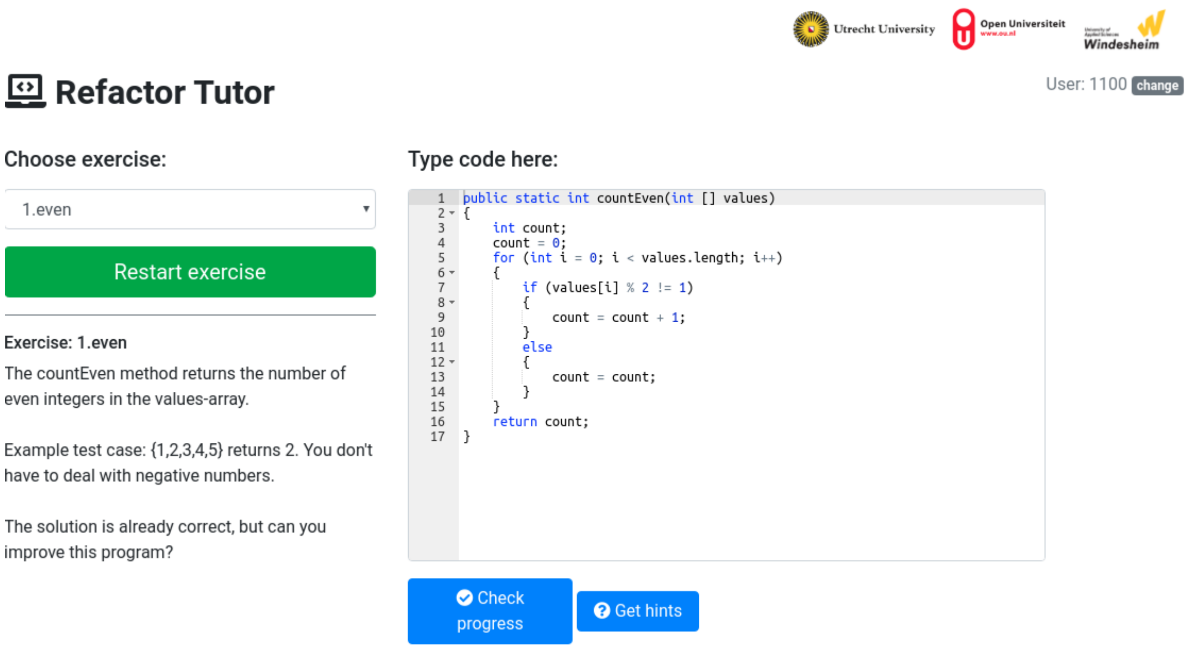
An example: if your savings are 1000 euros and the interest is 4%, it will take you 18 years to double your savings (then you'll have more than 2000 euros).

6.havethree: `haveThree(int [] nums)`

Given an array of ints, return true if the value 3 appears in the array exactly 3 times, and no 3's are next to each other.

Example test cases: haveThree with 3, 1, 3, 1, 3 returns true haveThree with 3, 1, 3, 3 returns false haveThree with 3, 4, 3, 3, 4 returns false

For this problem you have to write the code yourself. When the solution is correct and all test cases pass, you can check if there are hints to improve your program.



Exercise: 1.even

The countEven method returns the number of even integers in the values-array.

Example test case: {1,2,3,4,5} returns 2. You don't have to deal with negative numbers.

The solution is already correct, but can you improve this program?

```java
public static int countEven(int [] values)
{
    int count;
    count = 0;
    for (int i = 0; i < values.length; i++)
    {
        if (values[i] % 2 != 1)
        {
            count = count + 1;
        }
        else
        {
            count = count;
        }
    }
    return count;
}
```

Figure 2.1: Screenshot of the refactor tutor by Keuning et al. [2020]

2.3. Problem analysis

During refactoring, logic errors can quite easily occur. To illustrate the kinds of errors, let us look at Listing 2.1.

```java
public static int countEven(int [] values) {
    int count;
    count = 0;

    for (int i = 0; i < values.length; i++)   {
        if (values[i] % 2 != 1) {
            count = count + 1;
        } else {
            count = count;
        }
    }
    return count;
}
```

Listing 2.1: Count only the even numbers in a given array. A student might refactor the if-statement.

The exercise belonging to Listing 2.1 function is:

"The countEven method returns the number of even integers in the values-array. Example test case: {1,2,3,4,5} returns 2. You don't have to deal with negative numbers. The solution is already correct, but can you improve this program?"

The steps that can be taken to improve this program are:

1. Simplify the if statement: `values[i] % 2 == 0`: line 6
2. Remove useless else block (self-assignment): line 8-10
3. Rewrite calculation: `count ++`: line 7
4. Rewrite `for`-loop to `for-each`: `for(int value : values)`: line 5 and 6
5. Merge declaration and instantiation of `count` variable: `int count = 0;`: line 2 and 3

A student might refactor the if statement by changing the `!=` to a `==`, without changing the comparison value to 0. This creates the logical error that the wrong path of the if-statement is taken: only uneven numbers are counted. In the current version of the ITS, the message the student receives is:

```
Error:  Test case failed, calling countEven with [1, 2, 3, 4, 5]
should return 2, but your method returns 3
```

It only signals an error in the output, but it does not provide more insight in the origin of the error. This research is aimed at making the process that led up to the error insightful for the student. We will do this by writing multiple unit tests per exercise, specifically designed for observing various logic errors that may occur. These observations will be used to identify the underlying problem for an incorrect answer. For example, when the student makes the error as described above, it could display something along the lines of:

```
Error:  Test case failed, calling countEven with [1, 2, 3, 4, 5]
should return 2, but your method returns 3.  Is your check for
evenness correct?.
```

3
Research

In this research we will look at a way to help students better understand the problems they encounter when refactoring, within the context of solving exercises in a refactoring ITS. In this chapter we describe the questions we will answer during the research, how we will answer them and how we will verify the results we obtain.

3.1. Research questions

The main research question is “How to provide students feedback using unit tests for common logic errors when refactoring?". We have divided the main question into five research questions. The research will be focussed on practising refactoring within an Intelligent Tutoring System.

RQ.1 **What are common logic errors students make when they are refactoring?**

We can only provide feedback for common logic errors if we know what those errors are. This will be the first step of the research.

When looking at Listing 2.1 we can see – at least – one problem with the given code. One of the possible refactoring steps a student might take, is to rewrite the if-statement: it now has inequation `(!= 1)`, which can be rewritten to `== 0`. One error that might occur is that the student rewrites the equality operator to ==, but forgets to adjust the value from `1` to `0`. This will result in a faulty solution, as it will only count odd numbers.

We will perform a small literature review to make a list of common logic errors.

RQ.2 **Which common logic errors do students often make when they practice refactoring in an ITS?**

This research question concerns the identification of the logic errors that are most prevalent in our dataset.

We will examine the dataset as composed by a study of Keuning et al. [2020], and label the common logic errors made by the participants in that study. We will validate the labels by calculating the inter annotator agreement.

RQ.3 **How can we create unit tests that identify common logic errors to provide formative feedback?**

In this research question we will try to find a way to observe when students make common logic errors. We will make use of unit tests to implement buggy rules. This way, the observations can be implemented in the existing tutor, and provide more specific feedback to the student. We will be designing unit tests for the five existing exercises in the ITS as made by Keuning et al. [2020].

We will use binary decision trees to combine the results of different unit tests and recognise common logic errors.

When looking at Listing 2.1, a test case to observe the introduced logic error is that {1,2,3,4,5} returns 3.

RQ.4 **To what extent can we identify common logic errors using unit tests?**

In RQ.3, we automatically identified occurrences of the common logic errors, based on a subset of the dataset. Now we must validate if our method can indeed observe logic errors, by checking the performance on a held-out test set.

RQ.5 **How do teachers rate the feedback for common logic errors?**

To validate the feedback the tutor can provide based on automatically recognised common logic errors, we ask teachers to fill out a questionnaire about that feedback. This research question aims to find out if the feedback on certain common logic errors is in line with what teachers would give.

3.2. Research method and validation

To answer the questions posed, we have devised a method for each question. For each method we describe the steps to be taken, the output we expect and how we will validate the results.

3.2.1. RQ.1 - Finding common logic errors when refactoring

To find common logic errors, we will do a literature review and a log analysis. The literature review will be used to create an informed starting point for the log analysis.

Literature review

We study existing literature to find common logic errors that students make when refactoring. The articles as reviewed in Section 2.1 from Ettles et al. [2018], McCall and Kölling [2014], and the thesis by Kuah [2019] are a good starting point. All three of them provide insight into errors as made by students. These papers are the starting set for finding more articles.

Wohlin [2014] has created some excellent steps to systematically find more literature by snowballing. It is an iterative process, where one starts with a set of relevant articles, and use those to start snowballing. Backward snowballing is used to identify useful articles cited by the article (thus only finding articles backwards in time), whereas forward snowballing is looking who is citing the current article. For each found article one must determine whether it is relevant, and added to the set of articles. In a next iteration, the found articles can also be used for snowballing.

The criteria we use for selecting common logic errors from the literature are:

- Errors must not result in compiler errors
- Errors must be observable by unit tests (as opposed to stylistic errors)

Log analysis

In a paper written by Keuning et al. [2020], a study was done to investigate how a group of students worked with a refactoring ITS. The data from that study is available, and consists of a database with the logs generated by the students. These logs contain the submitted solutions, when a hint was asked and the code they submitted as a solution. This allows us to retrieve the incorrect solutions so we can determine what kind of common logic mistakes students make.

To identify common logic errors students make when using an ITS, the literature review will be an informed starting point to analyse this database. This will be done by a technique called log analysis. We will follow the following phases, as described by Jansen [2006]:

1. Examination

   The data has already been collected in the previous study. This phase will consist of a first examination of the available data. We will look at the different types of requests, the number of requests and what is inside each request.

2. Preparation

   We clean up the data, find out which information is useful and how we can query the database to answer questions for our analysis.

3. Analysis

   We will now analyse the submissions, and extract the errors made by the students.

The result of the literature review will be a list of common errors made by students. These errors are not specific to refactoring, and will serve to categorise the errors as found in the log analysis. The result from the log analysis will be a list of common logic errors made by students when refactoring.

We expect some overlap between the mistakes found in the literature review, and the log database. By comparing this overlap and analysing the discrepancies, we can make sure the results found for this research question are valid.

3.2.2. RQ.2 - Labelling common logic errors

To answer the question *Which common logic errors do students make often when they practice refactoring in an ITS?*, we must first identify the common logic errors present.

Working with the dataset from the study by Keuning et al. [2020], we will answer that question by labelling the errors present in the dataset. We will verify the outcome by cross validating a small sample by other annotators.

This research question is split into three phases: labelling of errors, dataset creation and validating the given labels by different annotators. The result of this research question will be the log database amended with labels for found errors, a training and a test set derived from the log database, and an inter annotator agreement.

Labelling of errors

In this phase, we label the errors that are present in the database. We do this by inspecting the submitted function, and identifying the error(s) present in said function. When identified, we update the row in the database with the label(s) identified in the function.

We make a hierarchy of labels: the errors we found in the literature review are used as a main category, and labels belong to one of those categories. A submission can contain multiple labels.

Splitting the data

We split the created dataset into two sets : one training set and one test set. The training set will be used to develop the unit tests against and to train the binary decision trees on the results of the unit tests. The test set will be used to verify if the binary decision trees identify the errors correctly.

These two datasets must both be representative of the entire population and be independent of each other. To this end, the sets must conform to the following rules:

1. Every exercise must be in both sets

2. Submissions from one student can at most be in one set

3. Duplicate entries from the same student are not allowed

The size of the datasets is influenced by the number of logic errors we want to recognise. The ratio between training set and test set is 80/20.

Inter annotator agreement

The labelling of the data is executed by the main author, which is a threat to validity. To mitigate this problem, we take a subset of the labelled instances, and have other researchers annotate the errors as well. To prevent contamination, we do not give the results of the first labelling action to the external annotators. We instruct the annotators on the labelling process by providing a document containing the labels, examples for each label, and an instruction on how to report the resulting labels.

We can then perform an analysis on that subset, and measure the level of agreement between different annotators. The simplest way according to Artstein [2017], is the *observed agreement*. This measure is the percentage of identical labels in the subset between all annotators. This metric gives insight in the reliability of the labels given by the main author.

3.2.3. RQ.3 - Observing common logic errors

The underlying assumption for this research is that we can observe common logic errors through the execution of unit tests. Unit tests have a predefined input, an expected output, and they execute a function-under-test (in our case the submission of a student). We can observe its actual output and compare it to the expected output. The outcome of this comparison is the result of the unit test.

This design has a couple of implications. The first implication is that the coupling between unit test and the function-under-test means that each unit test is linked to a specific exercise. The second implication is that we use dynamic analysis for the submitted code (i.e. the code is actually executed), and not static analysis (meaning the code is not executed). The submitted code must compile, otherwise the analysis cannot be done.

This research question consists of three phases:

1. Writing testbed software
2. Writing unit tests
3. Training of the models

Testbed software

The submissions in the database are strings containing Java code. To dynamically analyse them, tooling has to be written. This tooling will need to extract the functions from the database, compile them and load them in the Java Runtime. After that, it can be analysed by unit tests.

The result of the execution will be a `.csv`-file with all retrieved submissions and the results for the unit tests.

Writing unit tests

To observe logic errors one or more unit tests have to be written. The testbed as created in the previous phase, will determine the exact semantics of a unit test. However, the base premise still stands: we have some predefined input, an expected output and we compare the actual output against the expected output. In the case of Listing 3.1, the unit test can be:

```java
public static int countEven(int [] values) {
    int count;
    count = 0;

    for (int i = 0; i < values.length; i++)   {
        if (values[i] % 2 != 1) {
            count = count + 1;
        } else {
            count = count;
        }
    }
    return count;
}
```

Listing 3.1: Count only the even numbers in a given array. A student might refactor the if-statement.

```
assertThatNot( countEven([1,2,3,4,5]), 3);
```

This unit test will call the countEven method with an array containing the numbers 1 to 5, and checks if the output of the method is **not** equal to 3. When the students do change the operator from != to ==, but forget to change the value, the unit test fails (the result from the function will be 3). This result is then saved to the a `.csv`-file.

Train models

When all submissions have been analysed by the unit tests, the result is one large `.csv`-file. To analyse this file, we will use Orange, a tool written by Demšar et al. [2013]. Orange is a visual programming environment written in Python, to enable the user to implement various machine learning models. We will use Binary Decision Trees to create a model for identifying common logic errors.

We will use the training dataset (as created in Section 3.2.2) to train the model. The resulting model can be used to identify errors in new submissions: the behaviour of that submission will be observed with the unit tests, and the results from those unit tests will be analysed with the model from the Binary Decision Tree. The model will then give a prediction for what kind of common logic error this submission likely contains.

We chose a Binary Decision Tree to find any hidden links and dependencies between unit tests. The actual data in the training set will reveal those links and dependencies.

3.2.4. RQ.4 - Validation of RQ.3

This research question validates the Binary Decision Tree models, and to answer it, there must be some sort of indicator showing the effect. It gives us a measure on how accurate the models predict a common logic error, and how well it generalises beyond the training set.

Validate the model

To validate the models we take the test set. This test set will be used to let the unit test fire against, and let the models predict the likely error. This allows us to calculate the precision and recall of the proposed solution.

We have divided this into three consecutive parts:

Step 1:

We take the test set, and generate the output in the form of a csv file. This csv file contains the results of all unit tests.

Step 2:

This csv file will be used as an input for our models in Orange, written by Demšar et al. [2013]. Orange will be calculating the recall and precision of the models, and report it via confusion matrices.

Step 3:

Finally, we analyse the outcome from Step 2 and interpret the results.

These steps give us insight into how the created unit tests will “behave" in the real world and what their predictive value is. The result of this part will be two metrics, to indicate how well the designed unit tests can predict the identified errors and how many false positives will be found.

3.2.5. RQ.5 - Asking teachers to validate unit tests

For us to answer RQ.5, we will make a questionnaire. This questionnaire will contain a subset of the exercises, submission from a student and the feedback on that submission. This will be presented to teachers, so they can rate the feedback on the errors found.

The questions posed are of a qualitative nature, and are designed to provide insight into the appropriateness and meaningfulness of the provided feedback. The answers allow us to identify any missing elements, and to improve on the feedback.

We send this questionnaire to approximately five to ten teachers. This should allow us to retrieve enough information to validate the feedback, and rule out any possible misinterpretations made by the teachers.

Teachers are trained for their didactic skills, and thus are able to properly evaluate the feedback. We ask teachers in the software engineering department, so they have a firm grasp on the material and can evaluate the unit tests on logic and completeness.

3.3. Research contribution

This research contributes on a couple of areas. First, it provides insight in what kind of logical mistakes students make when they refactor functionally correct, but stylistically flawed code. This might provide some insight for teachers how to deal with these kind of problems. While there already is knowledge about common logic errors in a general programming context – as found by Ettles et al. [2018] – we make this knowledge more specific to a refactoring context.

Our second contribution is using unit tests to observe common logic errors in an ITS. While there are already tutoring systems that use unit tests for observing errors, this research focuses specifically on refactoring. Furthermore, it uses unit tests not only to identify errors, but also to identify nuances in the errors made. We hope this allows us to provide formative feedback on specific errors.

4
Results

4.1. Common logic errors when refactoring

To get an answer for research question 1, we study existing literature to find the current knowledge about common mistakes students make when programming. Then we do a log analysis on a database from a previous experiment with students refactoring. This log analysis will provide a list of errors specific to refactoring, while the errors from the literature review will serve as categories for the errors from the log analysis.

4.1.1. Literature review

For this literature review we look at common mistakes made by novice programmers. We define a common logic error as a computer program that does compile, but does not work as expected. By novice programmers we mean students, as this is the target for our research.

We started snowballing by using the articles by Ettles et al. [2018], McCall and Kölling [2014] and Kuah [2019], and from there compiled a list of literature, on which we snowballed further.

To provide a context, we first look at why student make errors when writing software. Then we look at the types of errors made by students. The result of this literature review is Table 4.1. The common errors from this table are used as categories in the log analysis.

Student errors

Learning to program is hard. According to Lahtinen et al. [2005], it deals with a lot of abstract concepts. Lahtinen et al. [2005] did a survey among teachers and students on the problems students encounter. They found that designing a program to solve a problem,dividing functionality into functions and debugging were found the most difficult issues. The difficult concepts are recursion, pointers and references, abstract data types, error handling and using language libraries. While the article by Lahtinen et al. [2005] does not have instances of common errors, it does provide some useful categories for identifying errors.

Besides the nature of programming, Gomes and Mendes [2007] also take into consideration the various circumstances: teaching methods, study methods, student abilities and attitudes, and the psychological effects. Gomes and Mendes [2007] see programming as a set of skills, instead of a single skill. Of these skills, they view problem solving as one of the most important abilities.

To categorise the common mistakes students make, Brown et al. [2014] did a large scale study on the Blackbox project. This is a project that collects data from the BlueJ IDE. The data in Blackbox contains some metadata for each user, start and end times of the programming session, and for each (compilation/edit/execution/etc.) event timestamps and relevant data. The granularity of this data makes it very usable for different kinds of research. Brown et al. [2014] focussed on compiler errors, and found that the top three compiler errors were: unknown variable, semicolon expected and unknown method.

In a later study, McCall and Kölling [2014] used the Blackbox data again. They tried to categorise the errors made by students who followed two introductory courses in Java programming. The research focused on determining a hierarchy of common errors. The errors retrieved from the dataset are all compile errors, and can be divided into syntactic, semantic and logic errors. Kölling et al. made a hierarchy for the categories, for example “variable not declared" is the parent category of “variable name written incorrectly". No details were given on the logic errors they identified. Our research focusses on dynamic analysis of code, which requires the code to compile. The resulting list of this study only contains compiler errors, so it was not incorporated in Table 4.1. However, it did provide some background and perspective on errors made by students.

Another study using the Blackbox data, this time by Altadmri and Brown [2015], reported specific mistakes. The mistakes ranged from syntax errors, type errors and other semantic errors. However, some errors could result in a logical error. Examples of these instances are confusing “short-circuit” evaluators (&& and ||) with & and |. These errors are included in Table 4.1.

Truong et al. [2004] created a static analysis tool to be used in ITSs and semi-automatic assessment tools. Using an Abstract Syntax Tree[1] it analyses different types of errors: poor programming practices and common logic errors. The common logic errors as described in this article are contained in Table 4.1, and the research into static analysis of common logic errors can be helpful for RQ.3.

Common logic errors

Ettles et al. [2018] researched the common logic errors made by students in their first year – after completing two programming courses. They tried to find what the most common errors are, and which are the most problematic for them to fix. Using the tool CodeWrite, they presented students with ten exercises to be programmed in C. The results from the compilable, but incorrect submissions were analysed. The results from the unit tests were evaluated and used to group similar solutions. Ettles et al. found that the most difficult and common misconception logic errors were: integer division results in an integer (not a double), uninitialised integers do not have a value (they assumed a 0), and off-by-one errors.

In his master thesis, Kuah [2019] describes some logic errors as made by first year students when solving programming exercises as taken from an exam. He shows that errors concerning string concatenation, mishandling of enum values and wrongfully assigning values to variables are among the most prevalent made errors. In Table 4.1 we have included his top five logic errors: these errors had the most occurrences and were best de-

[1]`https://en.wikipedia.org/wiki/Abstract_syntax_tree`

scribed in his thesis.

The PhD-thesis by Sorva [2012] includes a table with 162 misconceptions novice programmers have. These misconceptions are compiled from literature and exploratory research. The list is unordered and does not provide information about the commonness of the misconception. From this list we have selected eleven misconceptions that might be relevant to our research. We have excluded misconceptions that, for example, result in compiler errors, or are not relevant for refactoring one method.

Hristova et al. [2003] developed a tool to help students learn Java faster in introductory programming classes. The authors conducted research in often made mistakes, to create more readable errors messages reported by the tool. They surveyed college professors, teaching assistants and students to find programming mistakes students make with the language Java. For this article we also filtered out errors resulting in compiler errors. From the twenty errors reported by Hristova et al., we included six in Table 4.1.

Qian and Lehman [2017] did a study to various types of misconceptions students have when studying computer science. The authors identified that students have misconceptions and other difficulties about syntax, conceptual knowledge and strategic knowledge. For each type of misconception, they looked at what the contributing factors are. For strategic knowledge, it was found that students do not have complete and organised knowledge, lack strategies on how to solve a programming problem, and fail to reason at an abstract level. Qian and Lehman developed strategies to mitigate the found problems. For our research, we deemed the errors in strategic knowledge the most relevant for our study.

Result

From the papers studied, we compiled a list of 90 errors, mistakes and misconceptions. From this list we filtered out any irrelevant errors: errors resulting in compiler errors, errors that were too specific to the source article or too advanced for novice programmers. Lastly, we filtered out errors in constructs not present in the dataset for the log analysis. For example, the exercises only deal with primitives, so .equals and string operations are not relevant. The end result was a list of thirteen errors, as shown in Table 4.1. These errors are used to categorise errors as found in the log database.

Table 4.1: Filtered list of common errors made by students as found in literature.

ID	Error	Category name	Source
1	Division of two integers results in integer (no fraction)	Integer division	Ettles et al. [2018]
2	Off-by-one	Off-by-one	Ettles et al. [2018]
3	Confusing && and || with & and |	Short-circuit	Altadmri and Brown [2015]; Hristova et al. [2003]
4	Incorrect semicolon after if/for/while statement if (a == b); return 6;	Early semicolon	Altadmri and Brown [2015]; Hristova et al. [2003]
5	Not observing the bounds correctly (special case of Off-by-one)	Boundary	Ettles et al. [2018]; Qian and Lehman [2017]
6	If-if-else instead of if-else if else	Wrong if	Ettles et al. [2018]
7	Confusion between an array and its cell.	Array	Sorva [2012]
8	Not using a guard (prevent division by zero)	No guard	Qian and Lehman [2017]
9	Misconception: Both then and else branches are executed.	Both if-else	Sorva [2012]
10	Misconception: Using else is optional (the next statement is always the else branch)	Optional else	Sorva [2012]
11	Failing to check unexpected cases	Logic	Qian and Lehman [2017]
12	Wrong loop type for problem	Wrong loop	Qian and Lehman [2017]
13	Error handling is a difficult concept	Error handling	Lahtinen et al. [2005]

4.1.2. Log analysis

For this log analysis we use the data from a previous experiment by Keuning et al. [2020]. This experiment was done with 133 students using a refactoring tutor. This log analysis will produce a list with common logic errors students make when refactoring a function.

The log analysis will be done as described in Section 3.2.1. As reported by Keuning et al. [2020], there are six exercises in the dataset. The first five are in the form of: given a functionally correct function, can you improve it. However, the last exercise is different: it does not start with an implementation. This likely results in different errors, and are not comparable to the errors from the other exercises. Because of this, we exclude exercise six from the analysis.

Examination of database

The database consists of requests done by the refactor tutor between 26 September 2019 and 24 December 2019, while the experiment was conducted in the week of 14 October 2019. For a first filtering pass, we created a table containing only the data from that time period. All numbers reported here are from that time period.

The table with requests consists of twenty columns, containing metadata, the request itself, the input and the output. The most important metadata for this study consists of a timestamp, a sessionid, an identifier for the task, which service is called and the response time The input consists of a JSON object with all its parameters, as shown in Listing 4.1. The JSON object has the requested service (in this case `diagnoseR`), the parameters of the call (with the current exercise, the previous submission, the student id) and the new submission.

To get familiar with the data itself, we did some queries to get feeling for the amount of requests. Services are endpoints of the tutor, which can be called to validate an solution, get the number of hints remaining or the number of refactoring steps to be taken. A complete description of the endpoints can be found at the Ideas website [2]. In Table 4.2 we calculated the number of request per service. We are mostly interested in the diagnoseR service, as this provides the evaluation of the submitted solution of the service and reports any failing unit tests.

[2]`https://ideas.science.uu.nl/cgi-bin/rpt.cgi?input=%3Crequest%20service=%22servicelist%22%20encoding=%22html%22/%3E`

```
"event": <string>,     // currently empty
"id": <string>,        //currently always zero
"method": <string>,    // id of service
"params": [ [
   <string>,           // id of exercise
   <array>,            // empty for diagnoseR, used for expandHint
   <string>,           // previous submission
   <jsonobject>,       // Object with data for hints, empty for diagnoseR
   [
      <string>,        // user id
      "",
      ""
   ]],
   <string>            // New solution of student.
],
"source": <string>     // Calling user-agent

```

Listing 4.1: Definition of request body

Table 4.2: Requests done in experiment. Grouped per service type.

Service	**Requests**	**Explanation**
diagnoseR	10839	Diagnose current solution for refactoring
hintsremaining	4522	Returns the number of improvement left
allhints	1813	Returns a tree of feedback messages with increasing specificity
expandhint	1686	Expand on a hint, or ask for an alternative
example	1498	Returns a example expression that can be solved with an exercise
exerciselist	462	Returns all the exercises
stepsremaining	360	Returns number of steps still remain
onefirsttextr	82	Returns possible next step
allfirststextr	6	Returns all next steps

Similar to the input, the output column is also in JSON format. The JSON object contains if the exercise is ready, which exercise was processed, the processed code, any errors encountered (with accompanying information), the userid and the buggy rule which was triggered (in this case `improveevencheck`).

The service we are interested in, is the diagnoseR service. This is the service which evaluates the code from the student. The column `serviceinfo` provides us with information about the result of the service. The values that are present in the serviceinfo column for the diagnoseR service can be found in Table 4.3. Note that we have truncated the serviceinfo value to only show the main category of the serviceinfo. In the last column we provide a description of the serviceinfo. It becomes apparent that `NotEquivalent` is useful for this study.

Table 4.3: Requests per serviceinfo (only `service = diagnoseR`). First three rows are errors.

ServiceInfo	Requests	Meaning
NotEquivalent	1791	Test case is failing
Buggy	461	Reporting of buggy rules
SyntaxError	1180	Report compiler errors
Similar	4046	Submission is similar to starting point of exercise
Correct	1534	Submission is correct
Expected	709	Report expected next step

Preparation

The submitted functions are part of a JSON object in the column `input`. For easier evaluation of the data, we have created a new column named `submittedfunction`. For each failed unit test we have extracted the function from the JSON object, and updated this row with the offending function. We also added a column (`useridfunction`) with the function concatenated with the userid. This allows for easy counting of unique submissions per user. To find only relevant data, we must filter the dataset. We started with filtering the database for the given time period of the experiment, now we can extend those filters. The applied filters are:

1. Only use diagnoseR service: `service = ‘diagnoseR’`
2. Only failing unit tests: `serviceinfo like ‘NotEquivalent%’ or serviceinfo = ’buggycollapseif’`
3. Only unique submissions from the same user: `count(distinct(useridfunction));`

To get an overview of the errors, we wanted to know the amount of errors per exercise. This is shown Table 4.4. We have added the number of requests per exercise, to see what the relative amount of errors is.

Table 4.4: Per exercise the number of unique errors and requests, and the percentage of requests containing an error

Exercise	Errors	Requests	Percentage
1.even	177	5664	3.1%
2.sumvalues	489	4818	10.1%
3.oddsum	243	4439	5.5%
4.score	120	1475	8.1%
5.double	53	829	6.4%

Analysis

When looking at Table 4.4, we see that exercise two has the most relative errors. Upon inspection of the number of unique `serviceinfo`'s per exercise, we see a lot of errors only triggered a couple of times. To get a more manageable size table, we only look at errors encountered at least ten times. The result can be found in Table 4.5.

Table 4.5: Different errors per exercise.

Error	**Exercise**	**Errors**	**Subtotal**
No boolean condition in if	1.even	80	
countEven with [{1, 2, 3, 4, 5}] expected: 2, actual: 0	1.even	41	
countEven with [{1, 2, 3, 4, 5}] expected: 2, actual: 3	1.even	22	
Rest (errors with less than 11 occurrences)	1.even	34	
			177
sumValues with [{1, 2, 3, 4, -5}, true] expected: 10, actual: 5	2.sumvalues	302	
No boolean condition in if	2.sumvalues	50	
sumValues with [{1, 2, 3, 4, -5}, false] expected: 5, actual: 0	2.sumvalues	28	
sumValues with [{1, 2, 3, 4, -5}, true] expected: 10, actual: 15	2.sumvalues	26	
sumValues with [{1, 2, 3, 4, -5}, false] expected: 5, actual: 10	2.sumvalues	13	
sumValues with [{1, 2, 3, 4, -5}, true] expected: 10, actual: 4	2.sumvalues	11	
sumValues with [{1, 2, 3, 4, -5}, true] expected: 10, actual: 1	2.sumvalues	11	
sumValues with [{1, 2, 3, 4, -5}, true] expected: 10, actual: 0	2.sumvalues	11	
sumValues with [{1, 2, 3, 4, -5}, true] expected: 10, actual: -5	2.sumvalues	11	
Rest (errors with less than 11 occurrences)	2.sumvalues	26	
			489
oddSum with [{44, 12, 20, 1, -1, 3, 5, -1, 99, 4}] expected: 16, actual: 0	3.oddsum	38	
oddSum with [{44, 12, 20, 1, -1, 3, 5, -1, 99, 4}] expected: 16, actual: 20	3.oddsum	34	
oddSum with [{44, 12, 20, 1, -1, 3, 5, -1, 99, 4}] expected: 16, actual: 77	3.oddsum	28	
No boolean condition in if	3.oddsum	22	
oddSum with [{44, 12, 20, 1, -1, 3, 5, -1, 99, 4}] expected: 16, actual: 12	3.oddsum	18	
Test cases do not match the return type of the method	3.oddsum	17	
Your loop seems to be infinite, check the stop condition!	3.oddsum	16	
oddSum with [{44, 12, 20, 1, -1, 3, 5, -1, 99, 4}] expected: 16, actual: 15	3.oddsum	12	
Rest (errors with less than 11 occurrences)	3.oddsum	58	
			243
calculateScore with [2, 7] expected: 8, actual: 5	4.score	46	
calculateScore with [2, 3] expected: 5, actual: 8	4.score	19	
Rest (errors with less than 11 occurrences)	4.score	55	
			120
hasDoubled with [1000.0, 4] expected: 18, actual: 1	5.double	14	
hasDoubled with [1000.0, 4] expected: 18, actual: 0	5.double	12	
Rest (errors with less than 11 occurrences)	5.double	27	
			53
Total			**1082**

4.2. Labelling of common logic errors

In this section we will analyse the errors the students made in the study from Keuning et al. [2020], and we look at how well the labelling went by calculating the inter annotator agreement.

4.2.1. Labelling

Because the errors reported by the current version of the refactor tutor can be caused by multiple logical errors, we look at the functions as submitted by the students. We analyse what the error was and label the row in the database. The complete result of the labelling process can be found in Appendix A, but we have created a condensed version in Table 4.6. We excluded all errors with less than ten occurrences. Each label belongs to one main category, which can be found in Table 4.1 from the literature review.

In Table 4.7 we have counted the number of errors for each exercise per main category. We have added a row with a *Rest* category. Included here are the submissions with multiple labels, or main categories with less then 11 errors. This is visualized in Figure 4.1. In Table 4.8, we included the number of errors for each label. For readability, we have only included the label when the number of errors is larger or equals than ten, all labels with less than ten are grouped in the row *Rest*.

Table 4.7: For each exercise the number of errors per main category

Exercise	**Main category**	**Errors**
1.even	Array	80
1.even	Boundary	36
1.even	Logic	22
2.sumvalues	Logic	545
2.sumvalues	Array	56
2.sumvalues	Optional else	27
2.sumvalues	Boundary	12
3.oddsum	Logic	87
3.oddsum	Wrong loop	34
3.oddsum	Boundary	27
3.oddsum	Semantic	14
3.oddsum	Array	10
4.score	Logic	64
5.double	Logic	22
Rest		46
Total		**1082**

Table 4.8: For each main category the number of errors per label

Main category	**Label**	**Errors**
Logic	alwaysused	498
Array	foreachbutindex	147
Boundary	earlyexit	59
Logic	orinsteadofand	51
Logic	missingcase	52
Wrong loop	incorrectforeach	35
Logic	controlvarinverse	30
Logic	nostop	29
Optional else	onifsumdouble	27
Boundary	wrongincrement	27
Logic	incorrectlogicif	19
Logic	wrongvariableused	13
Logic	alsoaddedgecase	12
Off-by-one	indexoutofbounds	11
Logic	incorrectwhilecondition	10
Rest	-	62
Total	-	**1082**

Table 4.6: Labelled errors with explanation and the number of occurrences

Label	**Maincategory**	**Explanation**	**Errors**
alwaysused	Logic	No matter the outcome of the if, always add the value. Multiple variations: (a) one if with both then and else adding the number to the sum variable (b) if ((positivesOnly && i>0) || !positivesOnly), with the then also adding the number to the clause.	498
foreachbutindex	Array	Adjusted `for`-loop to `foreach`, but in if using the loop-variable as an index to retrieve a value from the array	147
earlyexit	Boundary	Stopping from the loop too early (after successful match, or after first iteration)	59
missingcase	Logic	Missing case: no else clause, or case from exercise premise forgotten	52
orinsteadofand	Logic	Wrong logical operand: && instead of ||, or vice versa	51
incorrectforeach	Wrong loop	Used foreach/while, when code needs an index, or when using a for-loop: skip nothing when indices have to be skipped	35
controlvarinverse	Logic	Stop variable initialized with true so it stops when false, but used inversely: only add to total when false. The opposite can also be true	30
nostop	Logic	No control variable present, used or manipulated to account for exercise parameters	29
onifsumdouble	Optional else	Always add the number to the sum, but again when the condition is true	27
wrongincrement	Boundary	Updated counter wrong: `count = count++;`	27
incorrectlogicif	Logic	Logic statement evaluates (almost) always to false	19
wrongvariableused	Logic	Checking wrong variable in if statement	13
alsoaddedgecase	Logic	Before stopping on found error, also added the edge case (-1) to the total	12
indexoutofbounds	Off-by-one	Retrieving value from array outside size of array, can be off-by-one	11
incorrectwhilecondition	Logic	While condition incorrect: only works on first iteration, or on wrong variables	10

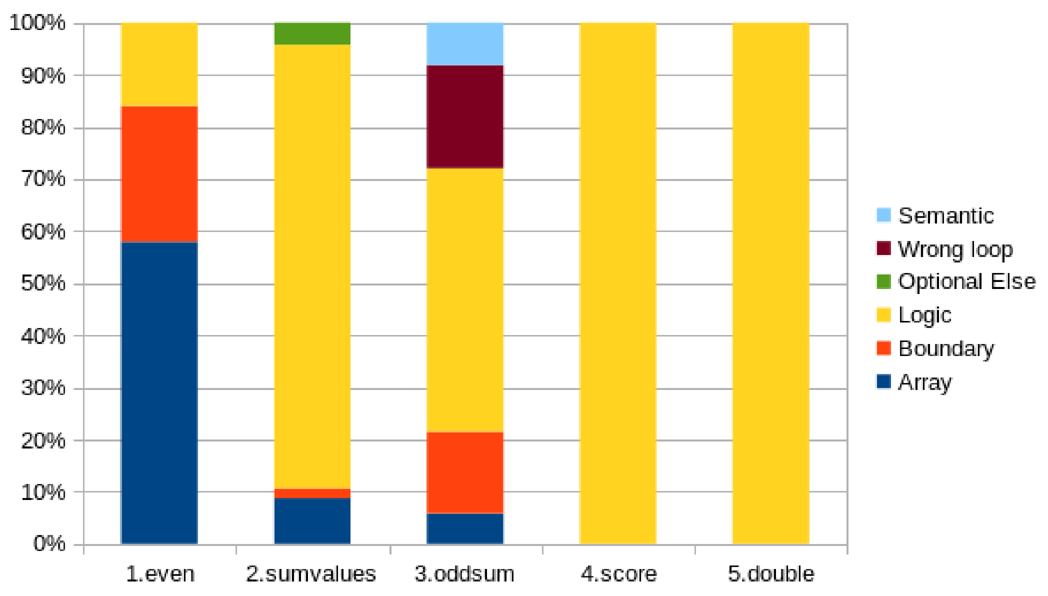


Figure 4.1: Distribution of main categories for each exercise. Data in Table 4.7.

The relationship between main category and label can be observed in Table 4.8. We can see that students make the most errors with the main categories Logic (69%) and Array (14%).

For the Logic category, the largest factor is the `alwaysused` label: regardless of the condition placed by the exercise, a computation is done (for example counting in exercise 1, or adding in exercise 2). Other large factors are using OR instead of AND, and not having an else clause. An example of this error can be found in Listing 4.2.

```java
public static int sumValues(int [] values, boolean positivesOnly)
{
    int sum = 0;
    for (int i : values)
    {
        if ((positivesOnly) && i>=0)
        {
            sum += i;
        }
        else
        {
            sum += i; // Same as in IF
        }
    }
    return sum;
}
```

Listing 4.2: Example for label alwaysused

The Array category is a category with only one label (`foreachbutindex`) as a child. However, it is the label with the second highest number of occurrences. This label is a misunderstanding of the for-each loop. An example for an instance is shown in Listing 4.4. We can see that the for-loop has been changed to a for-each, but the values-array is accessed with the current value of the iteration. Therefore, when passing the values `[4,1,3]` to the function, on the first iteration we get an IndexOutOfBounds (`values[4]` was requested, but the array was of length 3). This might be classified as an `indexoutofbounds` label, but `foreachbutindex` gives more nuance to the actual error (the error is not in wrongly checking the bounds, but misunderstanding the for-each construct).

We have added examples for the labels `earlyexit`, `foreachbutindex`, `missingcase`, `incorrectforeach` and `orinsteadofand` in respectively Listing 4.3, Listing 4.4, Listing 4.5, Listing 4.6 and Listing 4.7. We have added the complete list of examples in Appendix C.

```java
public static int countEven(int [] values)
{
    int count = 0;
    for (int i = 0; i < values.length; i++)
    {
        if (values[i] % 2 != 1)
        {
            count = count + 1;
        }
        else
        {
            return count; // Returns after first uneven
        }
    }
    return count;
}
```

Listing 4.3: Example for label earlyexit

```java
public static int countEven(int [] values)
{
    int count = 0;
    for (int i : values)
    {
        if (values[i] % 2 != 1) // i is the value, but used as an index
        {
            count++;
        }
        else
        {
            count = count;
        }
    }
    return count;
}
```

Listing 4.4: Example for label foreachbutindex

```java
public static int sumValues(int[] values, boolean positivesOnly)
{
   int sum = 0;
   for (int value: values)
   {
      if (positivesOnly == false){ // Only implements when parameter is
       false, other cases are missing
sum += value;
      }
   }
   return sum;
}
```

Listing 4.5: Example for label missingcase

```java
public static int oddSum(int [] values)
{
   int total = 0;
   boolean stop = false;
   for (int value : values) // foreach used, when exercise calls for
   for with index
   {
      if (stop == false)
      {
         if (value != -1)
         {
            total += value;
         }
         else
            if (value == -1)
            {
               stop = true;
            }
      }
   }
   return total;
}
```

Listing 4.6: Example for label incorrectforeach

```java
public static int calculateScore(int changes, int day)
{
    int score = 10;
    for (int i = 0; i < changes; i++)
    {
        score -= 1;
    }
    if (day != 6 || day != 7) // Always evaluates to true
    {
        score -= 3;
    }
    return score;
}
```

Listing 4.7: Example for label orinsteadofand

Interesting data

For one occurrence with value `NotEquivalent(No boolean condition in if)` we could not reproduce the error in the tool. When we entered the function, it gave a different error than reported in the database. We consider this row (timestamp: 2019-10-17 09:08:57.42462) a discrepancy, and labelled it according to the error reported in the tool. This can happen when a bug is fixed between the study and now.

The error *foreachbutindex* is found 147 times. Often this was during the refactoring of a simple for-loop to a for-each loop. Students did refactor the loop, but did not change how to access the value from the array: they replaced the index by the value from the for-each.

Some errors from the literature review – as noted in Table 4.1 – were not found in the log database, even though multiple articles reported on them. We pose the following explanations:

The errors *Integer division*, *Not using a guard to prevent division by zero* and *Error handling*, are errors that are out of scope from the exercises: there are no exercises where division and error handling is done.

The errors *If-if-else instead of if-else if else*, *Misconception: Both then and else branches are executed* and *Failing to check unexpected cases* are hard to analyse, as they require access to the thought process of students.

For the error *Confusing && and || with & and |*, we think that knowing the existence of the short-circuit operator is too advanced for students, while error *Failing to check unexpected cases* is prevented because of the example code already containing the initialisation of all variables.

4.2.2. Creating datasets

To create the datasets, we first assigned each userid a random number between 0 and 100. Based on that number, we put each userid into the test set (random number < 21) or the training set (random number > 20). This ensured that a user was only in one dataset.

We checked that the labels reported in Table 4.8 were present in both sets.

In the database there existed multiple duplicate requests for the same student, which were received shortly after each other – probably due to excessive clicking on the submit button. These duplicates are filtered out. The resulting datasets are described in Table 4.9 and in Table 4.10. To keep the tables legible, we have added an *other* category containing all errors with a low number of occurrences.

As we can see in Table 4.10, there are multiple errors made disproportionately often. The first six errors from the training set comprise 78% of the errors in that set. Therefore, we only focus on the errors: *alwaysused*, *foreachbutindex*, *earlyexit*, *orinsteadofand*, *missingcase* and *incorrectforeach*.

Table 4.9: Number of errors in the test set

Label	**Errors**
alwaysused	99
foreachbutindex	26
missingcase	16
earlyexit	12
controlvarinverse	8
incorrectforeach	7
nostop	6
onifsumdouble	6
orinsteadofand	6
wrongincrement	6
wrongvariableused	6
other (< 6 occurrences)	22
Total	220

Table 4.10: Number of errors in the training set

Label	**Errors**
alwaysused	399
foreachbutindex	120
earlyexit	47
orinsteadofand	45
missingcase	36
incorrectforeach	28
nostop	23
controlvarinverse	22
onifsumdouble	21
wrongincrement	21
incorrectlogicif	16
other (<11 occurences)	85
Total	863

4.2.3. INTER ANNOTATOR AGREEMENT

The labelling was done by the author, which is a threat to validity: the labelling – and successive training of the models – depended on one person. To mitigate this, we created a subset of the training set, and let two different researchers also label those submissions.

The labelling was based on a document supplied to the other researchers. This document contains the fifteen most used labels with one or more examples for each label, and one `other` category. The other category in the labelling process was chosen to limit the number of labels and simplify the labelling process for the other researchers. Besides the available labels, the document describes the environment and how to apply the labels.

The size of the subset was 77 submissions, containing all the exercises. We can find the results of the labelling process in Table 4.11. The number of instances where all three annotators were in total agreement was 48, which is 62.3%. The number of instances where at least one of the other annotators was in total agreement with the author is 57, which is 74%.

In Table 4.11, we can see that there are two possible types of findings why the agreement can go up: the partial match and other category.

Table 4.11: Agreement on submissions

Description	Annotator 1	Annotator 2	Explanation
Total checked	77	77	Total size of submission checked by other annotator
Direct match	52	53	Complete agreement on given label(s)
Other category	7	9	Other annotator assigned other category when label was not in labelling document
No match	17	9	There was no agreement on the label of the first author
Partial match	1	6	There was partial agreement: both annotators found a correct label

A partial match is when multiple labels were given by either annotator, but not all labels are being identified by the other annotator. For example: main author gives label A and B, where the Annotator 1 only gives label A. There is agreement on label A, but not on Label B. The reverse can also be true: Annotator 1 gives two labels, but the main author only one.

When the other researchers assigned the other category, we checked if the author assigned a label contained in the other category. We maintain the strict definition as described by Artstein [2017] and only count the identical labels.

The agreement number must be compared to a majority baseline: if the other labellers assigned only the majority class, how large the agreement would be. In this case, the majority class is the label `alwaysused` with 26 instances. Thus, the majority baseline is $\frac{26}{77} * 100 = 33.8\%$. The most strict calculation of the agreement (62.3%) is roughly two times better than the majority baseline. We can therefore conclude that the labelling that was done by the author is not dependent on luck, and to a high degree can be reproduced by others.

4.3. Writing unit tests to identify common logic errors

In this section, we analyse the results from RQ.3. We created some tooling to embed our unit tests in, written unit tests and trained six binary decision trees to classify common logic errors in submissions from students.

4.3.1. Testbed software

The submissions in the database are strings, which are not executable. We developed a program to extract the submissions, compile them and execute the unit tests. In Figure 4.2 an overview of the program is given. The processor reads submissions from the database and extracts the function from the student. It then repairs some minor defects (miscapitalized function name, removed static classifier), compiles the function into a .class file and loads the function in the Java Runtime. After that, the created unit tests are run, and the results of all the unit tests are saved to a csv file.

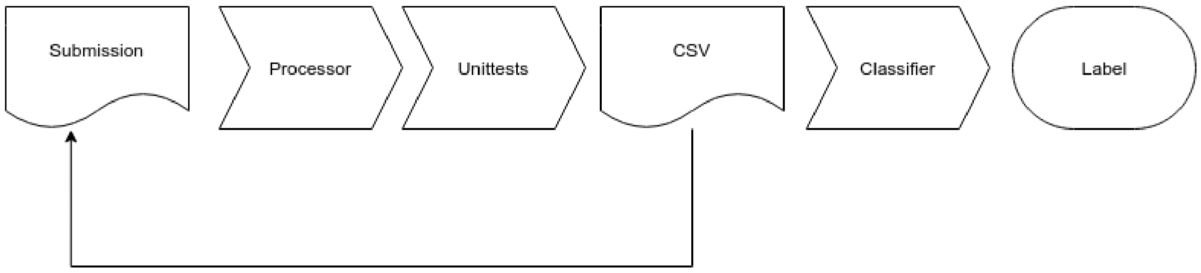


Figure 4.2: Design of testbed software

4.3.2. Writing unit tests

To observe errors in the datasets, we write several unit tests per label. In total, we created 27 unit tests. An example of a unit tests can be found in Listing 4.8. The first line of Listing 4.8 shows an annotation `@CommonLogicTest`. This annotation signifies that the following method is a unit test. It also provides information about the exercise which it acts upon.

Each unit tests receives a string containing the submission of the student as a parameter. Some input and corresponding expected output is prepared. When the method `executeSingle` is called, the submission of the student is called with the prepared input. The return value is compared against the expected value, and the result of that comparison is saved to the csv file via the method `addTestScore`. That method records the result of the unit test, together with its name.

4.3.3. Training the model

The results of the unit tests are written to a csv file. This file is read by a classifier as implemented in Orange: an application by Demšar et al. [2013] to develop algorithms using machine learning techniques. For the classifier algorithm we used a Binary Decision Tree, which we trained on the training data.

We created multiple models: one for each label. This allowed us to implement an algorithm to recognise multiple errors per submissions: the submission is checked for each label if it classifies as that label.

```java
@CommonLogicTest(functionNames = {"countEven"})
public void exitsAfterFirstIncorrect(String functionBody) throws
    Exception {
  Object[] input = {new int[]{2,2,2,1,2}};
  int expected = 3;
  Object result = null;
  result = executeSingle( input);
  addTestScore(result.equals(expected));
}
```

Listing 4.8: Example unit test

For this study, we focussed on the top six common logic errors: alwaysused, foreachbutindex, missingcase, incorrectforeach, earlyexit and orinsteadofand. Together these errors represent more than 75% of the total errors, as shown in Table 4.8.

4.4. Validate trained models

In this Section we evaluate the results from Section 4.3. By using the test set, we calculate metrics to analyse how the models perform. For all six common logic errors we show the overall performance, and we discuss several interesting cases.

4.4.1. Metrics

The metrics used to evaluate a trained model are precision and recall, F1 and the accuracy. To calculate these metrics we use the following terms:

TP True Positive: the model predicted correctly that the class applies

FP False Positive: the model wrongly predicted that the class applies

TN True Negative: the model predicted correctly that the class does not apply

FN False Negative: the model predicted wrongly that the class does not apply

Now we can calculate the metrics:

Precision
Ratio of correctly labelled errors: $\frac{tp}{tp+fp}$

Recall
Ratio of found errors: $\frac{tp}{tp+fn}$

F1
Harmonic mean of precision and recall: $\frac{2tp}{2tp+fp+fn}$

Accuracy
Ratio how well the model includes versus excludes errors: $\frac{tp+tn}{tp+tn+fp+fn}$

We also calculate the majority baseline of accuracy. This value represents the model when only the majority class is given to all the submissions. This allows us to place the accuracy in perspective: the accuracy must always be higher than the majority baseline, or else the model is worse at classifying when it should only assign the majority class.

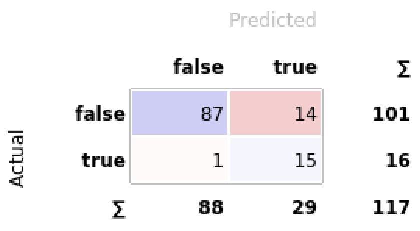
4.4.2. Analysis

When we look at Table 4.12, we see that all models perform above the majority baseline. This shows that the results are better than when we assign only the majority class to every submission.

Upon inspection, a few things stand out. First, the label `missingcase`. While the precision is high (0.924), the recall is quite lower: 0.872. Looking at Figure 4.3, we can see that this is because the model incorrectly classifies fourteen cases to be this error, when it should not have done that. While still high, when we compare the accuracy to the majority baseline, we note that they are close. For this label we can say that the task of classifying a `missingcase` label is hard. The closeness of the majority baseline and accuracy is also true for the labels `incorrectforeach`, `orinsteadofand` and `earlyexit`

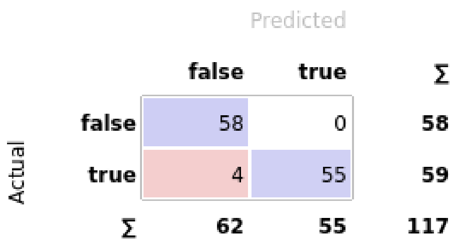
Table 4.12: For all labels, the metrics indicating how well the model performs

Label	Positive instances	Precision	Recall	F1	Majority baseline	Accuracy
missingcase	16	0.924	0.872	0.886	0.863	0.872
alwaysused	59	0.968	0.966	0.966	0.504	0.966
foreachbutindex	26	0.992	0.991	0.991	0.778	0.991
orinsteadofand	6	0.993	0.991	0.992	0.949	0.991
earlyexit	4	1.0	1.0	1.0	0.966	1.0
incorrectforeach	6	0.983	0.983	0.981	0.949	0.983



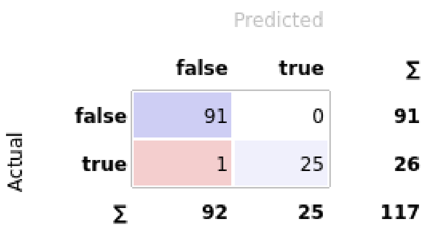
Actual \ Predicted	false	true	Σ
false	87	14	101
true	1	15	16
Σ	88	29	117

Figure 4.3: Confusion matrix of label missingcase



Actual \ Predicted	false	true	Σ
false	58	0	58
true	4	55	59
Σ	62	55	117

Figure 4.4: Confusion matrix of label alwaysused



Actual \ Predicted	false	true	Σ
false	91	0	91
true	1	25	26
Σ	92	25	117

Figure 4.5: Confusion matrix of label foreachbutindex

The second set of labels worth noting, are `earlyexit`, `orinsteadofand` and `incorrect-foreach`. The number of instances in the test set is very low for these labels, this makes the result statistically less relevant and we must dismiss these labels from the conclusion.

The last labels to note are `alwaysused` and `foreachbutindex`. Interesting here are the differences between the majority baseline and the accuracy: 0.462 and 0.213, respectively. This distance means the label determined by the model is not given by chance. The harmonic mean (F1-score) for both labels is high (0.966 and 0.991), and the number of positive instances is high enough to be relevant.

4.5. Validation by teachers

If the tutor can determine a common logic error in a student submission, it can provide feedback on how to solve that error. In this section we describe the feedback for the six logic errors for which we trained a model. Furthermore, we created a questionnaire to verify if the feedback is sufficiently informative to students. We asked twenty teachers to fill out that questionnaire, and in this section we report the results on that.

4.5.1. Feedback

We have created feedback on how to deal with the top six common logic errors. Multiple levels of feedback can be given, from very general ("Look at the loop.") to very specific ("You access the array by a value from that array instead of an index"). We chose for the middle ground here as it best shows the potential: it can recognize the error and provide feedback, but it does not give the answer straight away. The final implementation can opt to implement different levels of feedback. The audience of the feedback are students after one or two introductory programming courses.

The feedback for the six common logic errors are:

foreachbutindex

> The return value isn't correct. You changed the for loop to a for-each. This also changed the meaning of the loop variable, from an index to the actual value in the array. Did you forget to change how to access the value from the array?

earlyexit

> The return value isn't correct. It looks like not all values are considered. Did you return too early?

orinsteadofand

> The return value isn't correct. Look closely at the used logical operator(s). When do they evaluate to true?

alwaysused

> The return value isn't correct. It looks like everything is included in the result. Look at the conditions in the code. Do they reflect the exercise premise?

incorrectforeach

> The return value isn't correct. In your solution you used a for-each. Think about the construct of a regular for-loop.

missingcase

> The return value isn't correct. Read the exercise description again, and check if all the conditions are reflected in your solution.

4.5.2. Questionnaire

We created a questionnaire using Google Forms, as this allows for easy filling out the answers by respondents. The target audience were Computer Science/Software Engineering teachers in the Netherlands, so the language in which the questions are posed is Dutch. For reference, the complete questionnaire can be found in Appendix C.

In the questionnaire, we asked the teachers to rate the feedback given to three submissions by answering three questions about each submission. We gave the complete description of the exercise, the refactoring steps the students had to take, and a submission with a common logic error. We then explained the error the student made, and gave the corresponding feedback. The questions we asked were:

1. How much does the feedback correspond with the message you would give the student?
2. How do you rate the granularity of the feedback?
3. Do you have any remarks or suggestions about the given feedback?

Question one can be answered using a likert scale: a number from one to five, where one is “Not at all" and five is "Completely". In question two, there are three options:

- Too little: the student does not know anything
- Enough: the student can now figure out for themselves what the problem is and how to correct it
- Too much: the answer is being given

The last question did not use predefined answers, but was a textarea. Only question one and two are mandatory, question three could be skipped if the respondent so choses.

The labels for the submissions are, in order:

1. foreachbutindex
2. earlyexit
3. orinsteadofand

4.5.3. Results

We sent the questionnaire to twenty teachers from different Universities, of which eleven responded and answered the questions. In this section we will look at the given answers to question one and two, and which conclusions we can make. We use the open questions to give context and interpret the answers.

QUESTION 1: HOW MUCH DOES THE FEEDBACK CORRESPOND WITH THE MESSAGE YOU WOULD GIVE THE STUDENT?

In Figure 4.6, we show the answers to question one, for all three submissions. While there is some variation in how well the feedback is scored, we can conclude that to some degree we would give the same feedback as teachers would. The feedback for submission three is considered to be the best of all three: 82% of the teachers give it a four or five.

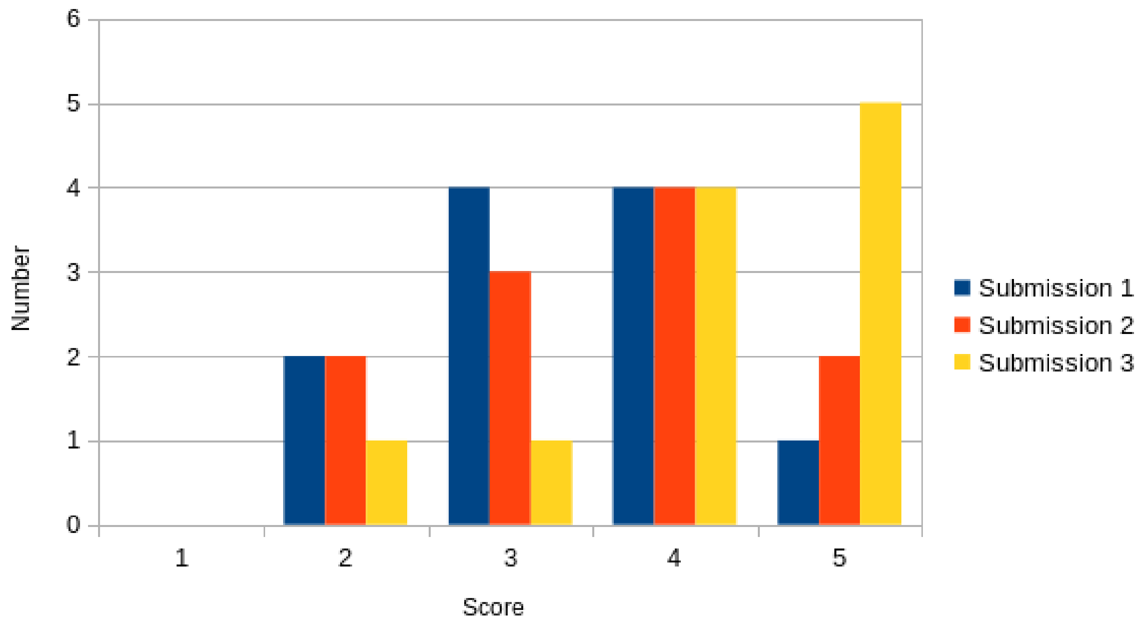


Figure 4.6: Combined answers for question 1: “How much does the feedback correspond with the message you would give the student?"

QUESTION 2: HOW DO YOU RATE THE GRANULARITY OF THE FEEDBACK?

Giving feedback is about balance: according to Shute [2008], too much detail can overwhelm the learner and promote superficial learning. This question aims to verify whether the granularity of the feedback is correct.

When we look at Figure 4.7, we see that for all the submissions the overall granularity of the feedback is considered to be good: not too little, and not too much. The only outlier here is submission one: five teachers think it gives away too much information.

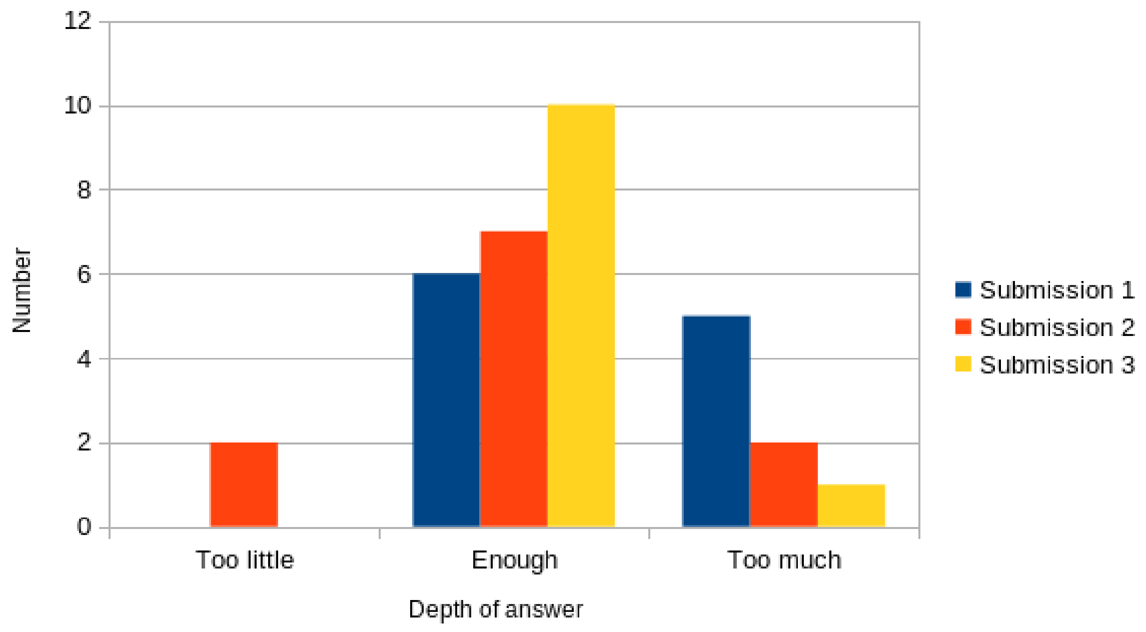


Figure 4.7: Combined answers for question 2: “How do you rate the granularity of the feedback?"

REMARKS

Several teachers noted that feedback must be attuned to the level of proficiency of the learner. While this was omitted from the questionnaire, the feedback was meant for students after one or two programming courses. We note that step-wise feedback might be implemented: starting with a very general remark, and finishing with the answer. We leave this for future work.

Others remarked upon the lack of context in the feedback. They suggest using the variable names used to provide more recognition. While this is certainly a good idea, the feedback as provided for the errors are independent of the exercise. Using a variable name would make the feedback specific for an exercise.

4.5.4. CONCLUSION

We have shown that the feedback in this section is of sufficient quality and granularity. The remarks obtained in the open questions do not show an inherent problem in the chosen strategy of using unit tests to recognize common logic errors to provide feedback, but merely provide suggestions in the implementation of the solution in the tutor.

4.6. Intermission - an example

To better explain the results from previous sections, we will inspect an example. This example will show for a given submission, how automatic feedback is provided. For this particular example, we look at the submission in Listing 4.10. The given code for exercise `4.score` can be found in Listing 4.9 and the description is as follows:

4.score

The calculateScore method calculates the score for a train trip. The highest score is 10. The score is based on the number of changes and the day of the week (Monday is 1, Sunday is 7).

Dutch Railways (NS) has designed the following calculation:
Base score: 10
For each change: -1
Trip on a weekday: -3

Example test case: for a trip with 2 changes on a Wednesday (day 3), calculateScore(2, 3) returns a score of 5 (10-2-3)

```java
public static int calculateScore(int changes, int day)
{
    int score = 10;
    for (int i = 0; i < changes; i++)
    {
        score = score - 1;
    }
    if (day == 6 || day == 7)
    {
        return score;
    }
    else
    {
        score = score - 3;
        return score;
    }
}
```

Listing 4.9: Exercise 4.score: code given to student to be improved.

When we inspect Listing 4.10,we can see that on line 5 the logical operand || is used. The result of the if-statement, is that this always evaluates to `true`. However, the body of the `if` is empty, but the `else` clause subtracts 3. The result of this construct is that the penalty for travelling on a weekday is never subtracted.

```java
public static int calculateScore(int changes, int day)
{
    int score = 10 - changes;

    if (day != 6 || day != 7)
    {

    }
    else
    {
        score -= 3;
    }
    return score;
}
```

Listing 4.10: Submission containing the error orinsteadofand

4.6.1. Running unit tests

To observe this incorrect behaviour, we have written three unit tests as shown in Listing 4.11. The first two, alwaysPenalty and wrongWeekendCheckInverse are for exercise 4. The third, boundaryCheck, is for exercise 3.oddsum.

alwaysPenalty

The first test, `alwaysPenalty`, checks if for all the days the weekday penalty is subtracted. This is done by calling the function with zero changes, and checking if for all the days the output of the function is equal to 7. We can see that this is **not** the case for Listing 4.11, so the result of this unit test is `false`.

wrongWeekendCheckInverse

The second, `wrongWeekendCheckInverse`, checks if the penalty for travelling on a weekday is never subtracted. As we saw above, this is the case. The result for this unit test is `true`.

boundaryCheck

The last unit test, `boundaryCheck`, is only for exercise 3, so it does not apply to this submission. This results in a `false` result for this unit test.

```java
public void alwaysPenalty(String functionBody) throws Exception {
    /**
     * Checks if function always adds a penalty for travelling, regardless of day
     */
    Object[] inputs = {1, 2, 3, 4, 5, 6, 7};
    List<Object> results = new ArrayList<>();
    for (Object input: inputs) {
        results.add(executeSingle(new Object[]{0,input}));
    }
    AtomicBoolean allWrong = new AtomicBoolean(true);
    results.forEach(res -> {
        if(!res.equals(7)){
            allWrong.set(false);
        }
    });
    addTestScore(allWrong.get()); // save result to csv file
}


 public void wrongWeekendCheckInverse(String functionBody) throws Exception {
     /**
      * Checks if function never adds a penalty for travelling on a weekday
      */
     Object[] inputs = {1, 2, 3, 4, 5};
     List<Object> results = new ArrayList<>();
     for (Object input: inputs) {
         results.add(executeSingle(new Object[]{0,input}));
     }
     AtomicBoolean allWrong = new AtomicBoolean(true);
     results.forEach(res -> {
         if(!res.equals(10)){
             allWrong.set(false);
         }
     });
     addTestScore(allWrong.get());
}

public void boundaryCheck(String functionBody) throws Exception {
     /**
      * Checks if function doesn't respect boundary because of using OR instead of AND
      */
     Object[] input = {new int[]{1, 1, 1, 1}};
     try{
         Object result = executeSingle(input);
     }catch(Exception e){
         if (((InvocationTargetException) e).getTargetException() instanceof ArrayIndexOutOfBoundsException) {
             addTestScore(true);
         }
     }
}
```

Listing 4.11: Three unit tests to check if the correct logical operand is used

4.6.2. Using the Binary Decision Tree

The trained model for the label `orinsteadofand` can be found in Figure 4.8. The description for the label `orinsteadofand` is: “Wrong logical operand: && instead of ||, or vice versa". Given the results from the unit tests as obtained above, we can walk through the tree.

We start at the top. The first unit test the model uses is `alwaysPenalty`. This result was `false`, so we take the left branch. We see that there is a 86.5% chance that the label `orinsteadofand` is not applicable to this submission.

The next unit test to inspect is `boundaryCheck`. The result was `false`, because it was not applicable for this exercise. Therefore, we take the left branch.

The last unit test we have to check is `wrongWeekendCheckInverse`. This result was `true`, so we can take the right branch, and we have reached a conclusion.

The label `orinsteadofand` is applicable to this submission. In this node we can see that there were only three submissions in the training set that take this path.

Now that we have determined the error in the given submission, we can present the student with feedback to tackle the common logic error. We can now give the following feedback:

The return value isn’t correct. Look closely at the used logical operator(s). When do they evaluate to true?

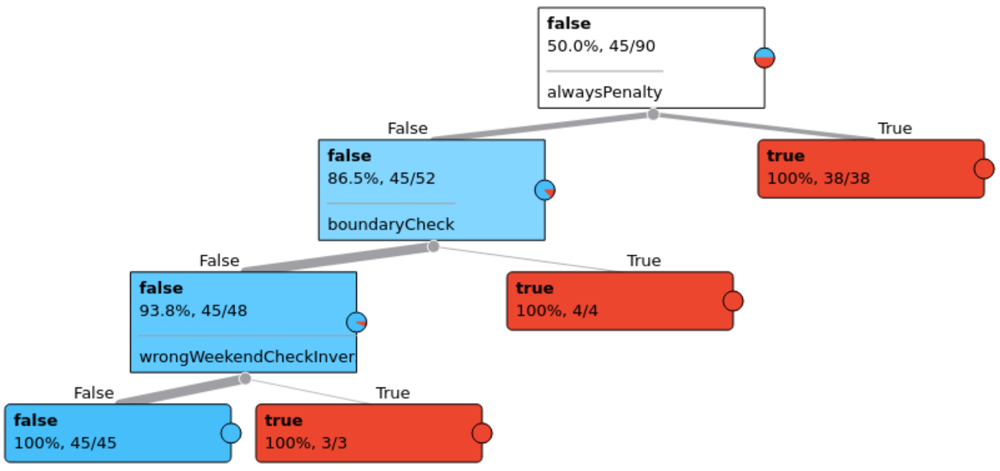


Figure 4.8: Trained model of Binary Decision Tree for label orinsteadofand

5 DISCUSSION

In this chapter we describe certain threats to validity: what factors compromise our research, and how did we mitigate them. We outline some solutions for problems we encountered that impacted the research, we discuss the limitations, and we provide some ideas for future research.

5.1. SOLUTIONS

In this section we give some insight in the decisions made in the setup of our research. We detail how we dealt with problems regarding the dataset, and choices made regarding the classification algorithm. We also shortly discuss some choices made for the inter annotator agreement.

5.1.1. IMBALANCED DATASET

A problem we encountered with the dataset is that it is quite imbalanced: of the six errors in this set, the first two labels – foreachbutindex and alwaysused – represent about 76%. This means that when we train one of the other labels – for example incorrectforeach, with 35 entries – it has relatively little data from which it can learn how to identify incorrectforeach. Even worse, when training for `incorrectforeach`, most of the data is about what is **not** `incorrectforeach`. To this end, we have made the dataset more balanced by using a technique called undersampling, as described by Kubat et al. [1997]. When training the model for an error, we limited the number of submissions not labelled for that error to the number of submissions that did satisfy that label. That ensured that the dataset was balanced, and the resulting model representative for the error it could recognise.

5.1.2. BINARY DECISION TREE

For recognizing common logic errors in student submissions, we used a binary decision tree as a classification algorithm. In this subsection we detail problems we encountered and how we dealt with them.

MODELS

When creating a model for the classification of a submission, we were faced with two choices on how to approach this: we could train a model per exercise or we could train a model per error. We choose the latter one. Here we set out to explain the rationale behind this decision.

Training a model per label has four advantages, but only one disadvantage.

1. Comparison of the trees between errors is easy
2. The whole point is identifying errors, so one model per label is more in line with our goal
3. It takes multiple labels per submission into account
4. All exercises are incorporated in the dataset

It has one disadvantage: the dataset is quite imbalanced. In the training set, the labels *alwaysused* and *foreachbutindex* represent around 60% of the set. The consequence is that when trying to train a different label, both *alwaysused* and *foreachbutindex* are overrepresented. We described the used solution to this problem in Section 5.1.1.

The other choice could be creating a model per exercise. This solution had only one advantage: comparing exercises is an easy task. This would make the difference and the effect of a unit test more explainable. However, the goal of the model is to identify an error – so we could provide more detailed feedback – and not to identify an error for a specific exercise. The disadvantages are:

1. Exercise four and five only have one common error, so no tree is needed
2. Multiple errors per submission is not possible

Upon inspection of the dataset, and in line with what one might expect, sometimes multiple errors were present in a submission. Overall it makes more sense to create a model per error: as shown above it has more advantages.

Features

The binary decision tree uses a csv-file as an input to train the model, and check if a submission contains a common logic error. The csv-file contains a submission per row, and on the columns are the features: information about that submission that together can determine if it contains a common logic error.

The obvious features of a submission are the results from the unit tests. A choice we made, was if the exerciseid should also be a feature. We decided against it, because it would make the exercise too important a factor in determining the error. We would over-fit the model to our current dataset, and the model would be harder to generalize to other exercises in the future. There is already an implicit connection between the error and the exercise: via the unit tests themself. A unit test is directly related to a function (the input parameters, and the return type are fixed), and thus to a exercise. While this tight coupling between unit tests and exercises might present a problem for expanding the tutor, we present some ideas for future work to remedy this in Section 5.3.1.

5.1.3. Inter annotator agreement

As Artstein [2017] notes, the observed agreement is a commonly used metric on how well labels have been given, but it has some drawbacks. It is easy to understand, but it says little about the annotation process and its reliability. He poses that using a coefficient from the kappa/alpha family is the accepted method for mitigating that problem.

While it is an interesting avenue to explore, our focus was on identifying common logic errors. The used observed agreement was sufficiently informative for our goal, but we leave this for future research.

5.2. Limitations

This research was set up to answer the question 'How to provide students feedback using unit tests for common logic errors when refactoring'. In this section we discuss the disadvantages and limitations of the chosen methods, and how we did mitigate them.

5.2.1. Literature research

We performed a literature research to determine common logic error students make when they learn to program. Due to the size of this part of the research, we did not do a systematic literature review. Consequently, we did not describe all the search terms we used to obtain the literature, which is a threat to verifiability. To mitigate this, we used the technique (reverse) snowballing. Using the same starting point, other studies can replicate the findings. To verify those findings further, we included the source of the article for each found common logic error.

5.2.2. Labelling

We determined which logic errors students make when they practise refactoring in the tutor. We did this by labelling the errors present in a database from a previous experiment. The labels were assigned by the main author, which is a threat to validity and reliability. We mitigated this by calculating the inter annotator agreement: two different researchers labelled a subset of the submissions, and we calculated the overlap between the other annotators and the author.

The wording of the labels used in this research is not always intuitive or reflective of the error it represents. Changing the labels could change the perception of the meaning of the label, and comparing the results between annotators would be less correct. This would, to some degree, invalidate the inter annotator agreement. Due to time constraints, we chose to not change them, but leave that for future research. The suggestions on the labels is included in Section 5.3.1.

5.2.3. Models

The validation for the binary decision trees was done by checking the models against a test set: a dataset not used in training the models, to prevent cross contamination of data and prevent over-fitting. For three of the common logic errors the number of instances in the test set was very low: four and six instances were in the test set. This low number caused a threat to validity if used, because the statistical relevance was too low. We did not use those errors in our interpretation of the results.

When validating the models, we used a smaller amount of errors in the test set for the labels `earlyexit` and `incorrectforeach` than reported earlier in this thesis (Section 4.2.2). When executing the unit tests, we discovered several submissions that were uncompilable, and thus could not be executed by unit tests. These submissions should not have been classified as such, but time did not permit to adjust all tables and datasets.

5.2.4. Questionnaire

We used a questionnaire to determine that the feedback provided by automatically recognized logic errors are deemed sufficient by teachers. This questionnaire contained three submissions from students. To reduce ambiguity for the reader – due to multiple errors in one submission, or not yet finished refactoring steps – we altered the submission. We removed errors that we did not want to focus on, or any skipped refactoring steps.

This might pose a threat to validity, as feedback on a submission is not given in isolation: in reality, a submission will contain multiple errors or skipped steps. However, we wanted to focus on providing one piece of feedback at the time: according to Shute [2008], presenting a student with all possible feedback at once, can cause cognitive overload.

The questionnaire was sent out to twenty different teachers, all within the Computer Science/Software Engineering domain. A threat to validity for this method is that the teachers asked all know the supervisor, which might influence the respondents to give favourable answers. We tried to minimize this by making the questionnaire completely anonymous, so they could give honest answers.

We sent the questionnaire to twenty teachers, of which eleven responded. While the response rate is quite high, the absolute number of respondents might be too low to generalise. To get a more conclusive answer to the question how the given feedback is rated by teachers, more research is needed.

5.3. Future work

After doing this research, we see some interesting areas to explore and expand on. We first give some hints on how one can extend this research, then we have some suggestions for different techniques on how to recognise common logic errors. Finally, we share some thoughts on the application of this research in a different area than (refactor) tutors.

5.3.1. Expanding this study

In Section 4.4, we encountered four common logic errors with too little data to validate the generated models. It would be interesting to conduct another experiment with more students to collect more data, so we can validate the models. This experiment can have multiple extensions on this research.

First, the labels can be enhanced to better reflect the meaning of it. We give some suggestions:

foreachbutindex

Error seems like an instance of a more general error, where a value is used as an index. That it occurs here in a foreach-loop is coincidence, but it can also happen in a different setting.

Suggestion: index-versus-value-at-index.

earlyexit
: Suggestion: returning-too-early.

alwaysused
: Explanation too specific for the exercises.

  Suggestion: no-difference-between-then-and-else

controlvarinverse
: The description is quite exercise specific.

  Suggestion: condition-inverse.

nostop
: That the variable is called stop is coincidence.

  Suggestion: unreachable-else

incorrectlogicif
: There is also condition-inverse, but that one is more specific. The explanation of the error is too exercise specific. There might be overlap with orinsteadofand.

  Suggestion: condition-incorrect.

Secondly, it might provide extra insight to trace the error to the refactor step the student tried to take. This could serve to give more context with the feedback.

Third, Keuning et al. [2017] provided a list of quality issues created by students, and also created the refactor tutor used in this thesis. It might be interesting to see if there is overlap between the quality issues reported in Keuning et al. [2017] and the list of common logic errors in this document.

Lastly, it might be interesting to create an algorithm that will determine the possible errors students will make in a new exercise, and automatically generate unit tests and accompanying feedback. This will make the creation of new exercises much easier, as unit tests and feedback does not have to be written for each new exercise. The bug database created in this research can be used as a starting point for this algorithm.

5.3.2. DIFFERENT TECHNIQUES

We chose a binary decision tree as a classification algorithm for recognising common logic errors. Several other classifiers exist for binary classification problems. It would be interesting to see how well other algorithms perform. For example, support vector machine or random forest are interesting avenues to investigate. Support vector machine is a supervised learning model that aims to classify based on the distance of a point on a hyperplane, while a random forest creates multiple binary decision trees and combines the outcome of the different trees. Preliminary testing suggests that using a random forest produces better results than a single binary decision tree.

Only checking compilable errors loses some recall: common logic errors resulting in non-compilable functions are not checked. A combination of static and dynamic analysis will be the strongest, and it might be interesting to investigate how well this combination works.

This study used unit tests that were manually created. This makes adding more exercises labour intensive. To mitigate this problem, it might prove beneficial to automatically generate unit tests. Using the errors described in this document, an algorithm might be developed to generate unit tests based on the exercise. This would allow the tutor to be expanded quite easily with new exercises. One can use property-based testing, as developed by Claessen and Hughes [2011].

5.3.3. Different areas of application

Another interesting path to investigate, is how well the results of this study generalize to other types of tutors. We can imagine that the unit tests created can also be incorporated to recognize common logic errors in general programming tutors.

It might be possible to use some findings from this thesis in a non-tutor environment. We can imagine that certain errors are always wrong, regardless of the context. For example, the error with label `foreachbutindex` is almost never correct. It can be advantageous for IDEs to incorporate these kinds of checks. A study whether these errors can be found using static analysis, or even generated unit tests, might be interesting.

6
Conclusion

This research set out to answer the question

> *"How to provide students feedback using unit tests for common logic errors when refactoring?"*

We did this by dividing this question into five sub-questions.

RQ.1 **What are common logic errors students make when they are refactoring?**

The first step was to identify the logic errors students make when refactoring. We compiled a list of twelve logic errors that were described in literature. These errors were not specific for refactoring, but more general. For example, off-by-one errors or the lack of a guard to prevent against a division by zero are commonly made mistakes. This list of errors, as found in Table 4.1, was used to define a hierarchy for the errors that are specific to refactoring.

RQ.2 **Which common logic errors do students make often when they practice refactoring in an ITS?**

Using the data from previous studies, we investigated what errors students actually made when refactoring. The errors in the dataset were labelled, and the datapoints were split into a training set and a test set. The labelling was done by the main author, and validated by two other authors. We found that the top two errors, `alwaysused` and `foreachbutindex`, were over-represented in the datasets. The other common logic errors students make when refactoring are `missingcase`, `orinsteadofand`, `earlyexit` and `incorrectforeach`.

RQ.3 **How can we create unit tests that identify common logic errors to provide formative feedback?**

We created unit tests to observe the top six common logic errors and we ran the unit tests against the training set. The results from the unit tests were used to train a binary decision tree, which is used to classify common logic errors. There was a need for a binary decision tree, because the relevance of all unit tests are not equal. Some tests are more indicative of a common logic error than others, or the combination of two unit tests point to a common logic error.

RQ.4 **To what extent can we identify common logic errors using unit tests?**

For two errors, we can say with a high degree of certainty if it is present in a student submission. For the other four errors, the test set was too small to validate the model. We suspect that, given enough data, we can also recognize the other four errors. When we can identify common logic errors made by students, we can provide feedback to help correct that mistake.

RQ.5 **How do teachers rate the feedback for common logic errors?**

For the top six common logic labels, we created feedback for students on how to mitigate the error. Using a questionnaire, we established that the message is consistent with the feedback that teachers would give. The granularity of the feedback that we propose, that is if the feedback provides the answer or only a hint, is rated as good by teachers.

We reported on some threats to validity of this research: a small sample size in the questionnaire or the exact wording of the labels that could be enhanced. Taken those threats into account, we can still say that – based on the results presented in this thesis – that for a subset of the found errors we can use unit tests to determine whether the error is present in a submission from the student. Furthermore, if we know that a submission contains such an error, we can provide feedback that is deemed relevant and of sufficient granularity. We believe this research provides material for new and interesting future research, such as the implementation of the label `foreachbutindex` into tools as PMD and SonarQube, or using different classification algorithms as random forests or Support Vector Machine.

Teachers have too little time to teach students refactoring. If the results of this research were to be implemented in the refactor tutor, students would need less help from the teacher. As such, refactoring could take a larger part of the curriculum in Software Engineering and Computer Science, without increasing the workload placed on the teachers.

Bibliography

Vincent Aleven. Help seeking and intelligent tutoring systems: Theoretical perspectives and a step towards theoretical integration. In *International handbook of metacognition and learning technologies*, pages 311–335. Springer, 2013.

Amjad Altadmri and Neil CC Brown. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 522–527, 2015.

Mauricio Finavaro Aniche and Marco Aurélio Gerosa. Most common mistakes in test-driven development practice: Results from an online survey with developers. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 469–478. IEEE, 2010.

Ron Artstein. Inter-annotator agreement. In *Handbook of linguistic annotation*, pages 297–313. Springer, 2017.

Barry W Boehm, John R Brown, and Mlity Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, pages 592–605, 1976.

Neil Christopher Charles Brown, Michael Kölling, Davin McCall, and Ian Utting. Blackbox: a large scale repository of novice programmers' activity. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 223–228, 2014.

Kim B Bruce. Five big open questions in computing education. *ACM Inroads*, 9(4):77–80, 2018.

Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. *ACM sigplan notices*, 46(4):53–64, 2011.

Janez Demšar, Tomaž Curk, Aleš Erjavec, Črt Gorup, Tomaž Hočevar, Mitar Milutinovič, Martin Možina, Matija Polajnar, Marko Toplak, Anže Starič, Miha Štajdohar, Lan Umek, Lan Žagar, Jure Žbontar, Marinka Žitnik, and Blaž Zupan. Orange: Data mining toolbox in python. *Journal of Machine Learning Research*, 14:2349–2353, 2013. URL `http://jmlr.org/papers/v14/demsar13a.html`.

Andrew Ettles, Andrew Luxton-Reilly, and Paul Denny. Common logic errors made by novice programmers. In *Proceedings of the 20th Australasian Computing Education Conference*, pages 83–89, 2018.

Gregor Fischer and Jürgen von Gudenberg. Improving the quality of programming education by online assessment. In *Proceedings of the 4th International Symposium on Principles and Practice of programming in Java*, pages 208–211. ACM, 2006. ISBN 9783939352051;3939352055;.

Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 2018.

Boby George and Laurie Williams. An initial investigation of test driven development in industry. In *Proceedings of the 2003 ACM symposium on Applied computing*, pages 1135–1139, 2003.

Anabela Gomes and António José Mendes. Learning to program-difficulties and solutions. In *International Conference on Engineering Education–ICEE*, volume 2007, 2007.

Amanda Harris, Victoria Bonnett, Rosemary Luckin, Nicola Yuill, and Katerina Avramides. Scaffolding effective help-seeking behaviour in mastery and performance oriented learners. In *AIED*, pages 425–432, 2009.

Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. Identifying and correcting java programming errors for introductory computer science students. *ACM SIGCSE Bulletin*, 35(1):153–156, 2003.

Bernard J Jansen. Search log analysis: What it is, what's been done, how to do it. *Library & information science research*, 28(3):407–432, 2006.

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*, pages 110–115, 2017.

Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education (TOCE)*, 19(1):1–43, 2018.

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*, pages 1–10, 2020.

Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. A tutoring system to learn code refactoring. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, pages 1–7, 2021.

Diana Kirk, Tyne Crow, Andrew Luxton-Reilly, and Ewan Tempero. On assuring learning about code quality. In *Proceedings of the Twenty-Second Australasian Computing Education Conference*, pages 86–94, 2020.

W.C. Kuah. Augmenting CBM to generate constraints and reflection-based feedback for logic errors in programming solutions. Master's thesis, Open Universiteit, the Netherlands, 2019.

Miroslav Kubat, Stan Matwin, et al. Addressing the curse of imbalanced training sets: one-sided selection. In *Icml*, volume 97, pages 179–186, 1997.

Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. *ACM sigcse bulletin*, 37(3):14–18, 2005.

Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4): 308–320, 1976.

Davin McCall and Michael Kölling. Meaningful categorisation of novice programmer errors. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*, pages 1–8. IEEE, 2014.

Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139, 2004.

Antonija Mitrovic, Kenneth R Koedinger, and Brent Martin. A comparative analysis of cognitive tutoring and constraint-based modeling. In *International Conference on User Modeling*, pages 313–322. Springer, 2003.

Sreerama K Murthy. Automatic construction of decision trees from data: A multi-disciplinary survey. *Data mining and knowledge discovery*, 2(4):345–389, 1998.

Yizhou Qian and James Lehman. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education (TOCE)*, 18(1):1–24, 2017.

Valerie J Shute. Focus on formative feedback. *Review of educational research*, 78(1):153–189, 2008.

Juha Sorva. *Visual Program Simulation in Introductory Programming Education.* PhD thesis, 05 2012.

Nghi Truong, Paul Roe, and Peter Bancroft. Static analysis of students' Java programs. In *Proceedings of the Sixth Australasian Conference on Computing Education-Volume 30*, pages 317–325. Citeseer, 2004.

Kurt VanLehn. The behavior of tutoring systems. *International journal of artificial intelligence in education*, 16(3):227–265, 2006.

Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, pages 1–10, 2014.

APPENDICES

A. ERRORS

Label	Maincategory	Explanation	Occurrences
alwaysused	Logic	No matter the outcome of the if, always add the value. Multiple variations: (a) one if with both then and else adding the number to the sum variable (b) if ((positivesOnly && i>0) || !positivesOnly), with the then also adding the number to the clause.	498
foreachbutindex	Array	Adjusted `for`-loop to `foreach`, but in if using the loop-variable as an index to retrieve a value from the array	147
earlyexit	Boundary	Stopping from the loop too early (after successful match, or after first iteration)	59
missingcase	Logic	Missing case: no else clause, or case from exercise premise forgotten	52
orinsteadofand	Logic	Wrong logical operand: && instead of ||, or vice versa	51
incorrectforeach	Wrong loop	Used foreach/while, when code needs an index, or when using a for-loop: skip nothing when indices have to be skipped	35
controlvarinverse	Logic	Stop variable initialized with true so it stops when false, but used inversely: only add to total when false. The opposite can also be true	30
nostop	Logic	No control variable present, used or manipulated to account for exercise parameters	29

onifsumdouble	Optional else	Always add the number to the sum, but again when the condition is true	27
wrongincrement	Boundary	Updated counter wrong: `count = count++;`	27
incorrectlogicif	Logic	Logic statement evaluates (almost) always to false	19
wrongvariableused	Logic	Checking wrong variable in if statement	13
alsoaddedgecase	Logic	Before stopping on found error, also added the edgecase (-1) to the total	12
indexoutofbounds	Off-by-one	Retrieving value from array outside size of array, can be off-by-one	11
incorrectwhilecondition	Logic	While condition incorrect: only works on first iteration, or on wrong variables	10
doubleloop	Logic	Double loop, resulting in infinite loop	8
positivecheckincorrect	Logic	Mistaken boolean logic: if (values[i] % 2 == 0) && value <0	8
modulowrong	Logic	Modulo operation checks for wrong value: `if (values[i] % 2 == 1)`	7
assignmentwrong	Semantic	Self-assign instead of incrementing, or signs in wrong order: count =+ 1;	7
forloopincrement	Wrong loop	Incrementing for-loop counter wrong: `for (int i = 0; i <values.length; i += 2),` resulting in a possible out-of-bounds	6
toocomplicatedfor	Logic	Complicated definition of for-loop: `for (int i = 1; i <values.length && values[i] == -1; i += 2)`	4
counterinitwrong	Off-by-one	Start counting with 1	3
emptyif	Logic	No statements to be executed when if evaluates to true	3
noincrement	Logic	Not calculating the number of years	3

countinsteadofsum	Logic	Add 1 instead of the value: sum += 1	2
ifelsesame	Logic	The calculations are the same for the stop variable: everything is added	2
inconsistentcontrolvar	Logic	Semantic of control variable stop used inconsistently: initialised with true and adjusted if statement to reflect, but when `-1` is found, set to true again instead of false	2
noloop	Logic	No loop used when needed	2
arraywrongindexed	Array	Use wrong variable as index: `if (values[count] % 2 == 0)`	1
continuewhennotstop	Logic	Misuse of continue directive	1
divisioninsteadofmodule	Logic	if (values[i] / 2 == 0)	1
ifwithoutaccolades		if (array[i] == -1) stop = true; return total; After setting stop to true, immediately return. Probably error (code was indented)	1
incrementwithitself	Logic	`count += count ;`	1
logicstatementsswitched	Logic	Multiple logic statements switched: value for one used by the other	1
modulooutofbounds	Boundary	Modulo operation that could never evaluate to true: `if (values[i] % 2 == 2)`	1

Table 6.1: Labelled errors with explanation and the number of occurrences

B. Exercises

```java
public static int countEven(int [] values)
{
    int count;
    count = 0;
    for (int i = 0; i < values.length; i++)
    {
        if (values[i] % 2 != 1)
        {
            count = count + 1;
        }
        else
        {
            count = count;
        }
    }
    return count;
}
```

Listing 6.1: Exercise 1.even: The countEven method returns the number of even integers in the values-array.

```java
public static int sumValues(int [] values, boolean positivesOnly)
{
    int sum = 0;
    for (int i = 0; i < values.length; i++)
    {
        if (positivesOnly == true)
        {
            if (values[i] >= 0)
            {
                sum += values[i];
            }
        }
        else
        {
            sum += values[i];
        }
    }
    return sum;
}
```

Listing 6.2: Exercise 2.sumvalues: The sumValues method adds up all numbers from the values-array, or only the positive numbers if the positivesOnly boolean parameter is set to true.

Listing 6.3: Exercise 3.oddsum: The method oddSum returns the sum of all numbers at an odd index in the array parameter, until the number -1 is seen at an odd index.

```java
public static int oddSum(int [] array)
{
    int total = 0;
    boolean stop = false;
    for (int i = 1; i < array.length; i = i + 2)
    {
        if (stop == false)
        {
            if (array[i] != -1)
            {
                total += array[i];
            }
            else
                if (array[i] == -1)
                {
                    stop = true;
                }
        }
        else
        {
            total = total;
        }
    }
    return total;
}
```

```
public static int calculateScore(int changes, int day)
{
    int score = 10;
    for (int i = 0; i < changes; i++)
    {
        score = score - 1;
    }
    if (day == 6 || day == 7)
    {
        return score;
    }
    else
    {
        score = score - 3;
        return score;
    }
}
```

Listing 6.4: Exercise 4.score: The calculateScore method calculates the score for a train trip. The highest score is 10. The score is based on the number of changes and the day of the week (Monday is 1, Sunday is 7). Dutch Railways (NS) has designed the following calculation: Base score: 10 For each change: -1 Trip on a weekday: -3

```
public static int hasDoubled(double savings, int interest)
{
    double target = 2 * savings;
    int years;
    for (years = 0; ; )
    {
        if (target > savings)
        {
            savings *= interest / 100.0 + 1;
            years++;
        }
        else
            if (target <= savings)
            {
                break;
            }
    }
    return years;
}
```

Listing 6.5: Exercise 5.double: Write a program that calculates in how many years your savings have doubled with the given interest (as a percentage).

```
public static boolean haveThree(int [] nums)
{
    return true;
}
```

Listing 6.6: Exercise 6.havethree: Given an array of ints, return true if the value 3 appears in the array exactly 3 times, and no 3's are next to each other. No correct code given. Not used in this study.

C. Examples

```java
public static int sumValues(int [] values, boolean positivesOnly)
{
    int sum = 0;
    for (int i : values)
    {
        if ((positivesOnly) && i>=0)
        {
            sum += i;
        }
        else
        {
            sum += i; // Same as in IF
        }
    }
    return sum;
}
```

Listing 6.7: alwaysused

```java
public static int countEven(int [] values)
{
    int count = 0;
    for (int i : values)
    {
        if (values[i] % 2 != 1) // i is the value, but used as an index
        {
            count++;
        }
        else
        {
            count = count;
        }
    }
    return count;
}
```

Listing 6.8: foreachbutindex

```java
public static int calculateScore(int changes, int day)
{
    int score = 10;
    for (int i = 0; i < changes; i++)
    {
        score -= 1;
    }
    if (day != 6 || day != 7) // Always evaluates to true
    {
        score -= 3;
    }
    return score;
}
```

Listing 6.9: orinsteadofand

```java
public static int countEven(int[] values)
{
    int count = 0;
    for (int i = 0; i < values.length; i++)
    {
        if (values[i] % 2 != 1)
        {
            count = count + 1;
        }
        else
        {
            return count; // Returns after first uneven
        }
    }
    return count;
}
```

Listing 6.10: earlyexit

```java
public static int sumValues(int[] values, boolean positivesOnly)
{
    int sum = 0;
    for (int i = 0; i < values.length; i++)
    {
        if (positivesOnly)
        {
            if (values[i] >= 0)
            {
                sum += values[i]; // Only implements when value is positive
    , other cases are missing
            }
        }
    }
    return sum;
}
```

Listing 6.11: missingcase

```java
public static int sumValues(int[] values, boolean positivesOnly)
{
    int sum = 0;
    for (int value: values)
    {
        if (positivesOnly == false){ // Only implements when parameter is false, other cases are missing
            sum += value;
        }
    }
    return sum;
}
```

Listing 6.12: missingcase

```java
public static int calculateScore(int changes, int day)
{
   int score = 10;
   for (int i = 0; i < changes; i++)
   {
      score -= 1;
   }
   if (day == 6) // Only implements case for saturday, sunday is considered a weekday.
   {
      return score;
   }
   else
   {
      score -= 3;
      return score;
   }
}
```

Listing 6.13: missingcase

```java
public static int calculateScore(int changes, int day)
{
   int score = 10;
   if(day <= 5){ // multiple cases from exercise premise missing
      score -= changes;
   }
   return score;
}
```

Listing 6.14: missingcase

```java
public static int oddSum(int [] values)
{
    int total = 0;
    boolean stop = false;
    for (int value : values) // foreach used, when exercise calls for
    for with index
    {
        if (stop == false)
        {
            if (value != -1)
            {
                total += value;
            }
            else
                if (value == -1)
                {
                    stop = true;
                }
        }
    }
    return total;
}
```

Listing 6.15: incorrectforeach

```java
public static int oddSum(int [] array)
{
    int total = 0;
    boolean stop = false;
    for (int i = 1; i < array.length; i = i + 2)
    {
        if (stop) // will never enter, should be initialised to true
        {
            if (array[i] != -1)
            {
                total += array[i];
            }
            else if (array[i] == -1)
            {
                total = true; // inversed again
            }
        }
    }
    return total;
}
```

Listing 6.16: controlvarinverse

```java
public static int oddSum(int [] array)
{
   int total = 0;
   boolean stop = false;
   for (int i = 1; i < array.length; i += 2)
   {
      if (!stop)
      {
         if (array[i] != -1)
         {
            total += array[i];
         }
      } else if(array[i] == -1) {
         stop = true; // would never reach
      }

   }
   return total;
}
```

Listing 6.17: nostop

```java
public static int countEven(int [] values)
{
   int count;
   count = 0;
   for (int i = 0; i < values.length; i++)
   {
      if (values[i] % 2 == 0)
      {
         count = count ++;// would not update count variable
      }
      else
      {
         count = count;
      }
   }
   return count;
}
```

Listing 6.18: wrongincrement

```java
public static int sumValues(int [] values, boolean positivesOnly)
{
    int sum = 0;
    for (int i : values)
    {
        if (positivesOnly)
        {
            if (i >= 0)
            {
                sum += i;
            }
            sum += i; // always added, resulting in added to sum twice
    when positive
        }
    }
    return sum;
}
```

Listing 6.19: onifsumdouble

```java
public static int oddSum(int [] array)
{
    int total = 0;
    boolean stop = false;
    for (int i = 1; i < array.length; i = i + 2)
    {
        if (!stop)
        {
            if (array[i] == -1)
            {
                stop = true;
            }

            total += array[i]; // when array[i] is -1, the value -1 is
    also added
        }

    }
    return total;
}
```

Listing 6.20: alsoaddedgecase

```java
public static int calculateScore(int changes, int day)
{
    int score = 10;
    for (int i = 0; i < changes; i++)
    {
        score -= 1;
    }
    if (day <= 1 && day <= 5){ // incorrect, only true for day = 1
        score -= 3;
    }
    return score;
}
```

Listing 6.21: incorrectlogicif

```java
public static int hasDoubled(double savings, int interest)
{
    double target = savings * 2;
    int years = 0;
    while (years == 0) // wrong variable used in while condition
    {
        if (target > savings)
        {
            savings *= interest / 100.0 + 1;
            years++;
        }
        else
        if (target <= savings)
        {
            break;
        }
    }
    return years;
}
```

Listing 6.22: incorrectwhilecondition

```java
public static int countEven(int[] values)
{
    int count = 0;
    for (int i = 0; i < values.length; i++)
    {
        if (i % 2 != 1) // i is the index here
        {
            count = count + 1;
        }
    }
    return count;
}
```

Listing 6.23: wrongvariableused

```java
public static int oddSum(int [] array)
{
   int total = 0;
   boolean stop = false;
   for (int i = 0; !stop && i < array.length; i = i + 2) // first i = i + 2 allows for an out of bounds
   {
      if (array[i + 1] != -1) // Second time: i + 1  allows for an out of bounds
      {
         total += array[i + 1];
      }
      else
      {
         stop = true;
      }
   }
   return total;
}
```

Listing 6.24: indexoutofbounds

```java
public static int countEven(int [] values)
{
   int count = 0;;
   for(int num : values)
   {
      if(count %2 == 0) // count used for the even check, should be 'num'
      {
      count++;
      }
   }
   return count;
}
```

Listing 6.25: wrongvariableused

D. Questionnaire

Feedback op veelgemaakte fouten

Beste docent,

Momenteel ben ik, Meine Toonen, bezig met mijn afstudeeronderzoek voor mijn master aan de Open Universiteit.
Binnen de Open Universiteit worden verschillende tutoren ontwikkeld: (web) applicaties die een student kunnen helpen een bepaalde vaardigheid aan te leren, zonder (al te veel) tussenkomst van docenten.

Mijn onderzoek gaat over zo'n tutor: de refactor-tutor. In deze tutor kunnen studenten oefenen met het refactoren van code. Ik richt me op het herkennen van veelgemaakte fouten. We denken dat studenten veel dezelfde fouten maken, wat betekent dat we – als we deze fouten herkennen – we hier gericht feedback op kunnen geven.

De eerste resultaten zijn veelbelovend: voor een zestal typen fouten hebben we een manier gevonden om automatisch vast te stellen of ze gemaakt worden. We kunnen hiervoor dus feedback geven aan de student die hier wat aan heeft.

De feedback die gegeven wordt, heb ik zelf geschreven. Via deze vragenlijst wil ik vaststellen of dit het soort feedback is dat jullie als vakinhoudelijke didactici ook zouden geven.

De vragenlijst bestaat uit drie onderdelen, en zal ongeveer tien minuten in beslag nemen. Bij elke onderdeel zal de opgave worden gegeven zoals de leerling deze krijgt. Bij de opgave zullen ook een of meerdere inzendingen staan die een fout bevatten. Aan u de vraag om de feedback die het systeem gegenereerd te evalueren en aan te geven of u het ermee eens bent. Bij het analyseren van de feedback wil ik u vragen niet naar de exacte bewoording te kijken, maar om de achterliggende boodschap te beoordelen.

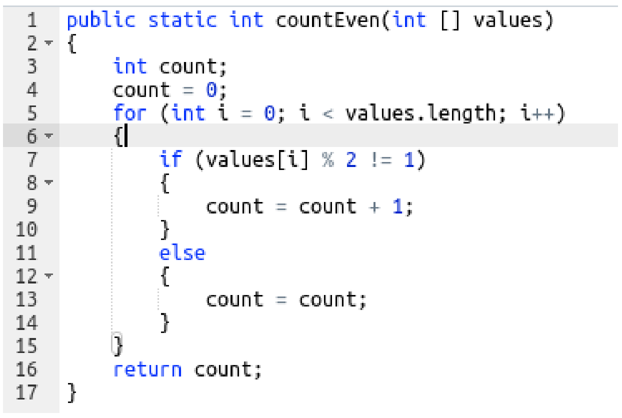
Alvast hartelijk bedankt!

* Required

Vraag 1

De eerste twee vragen gaan over de volgende opgave:

The countEven method returns the number of even integers in the values-array. Example test case: {1,2,3,4,5} returns 2. You don't have to deal with negative numbers. The solution is already correct, but can you improve this program?



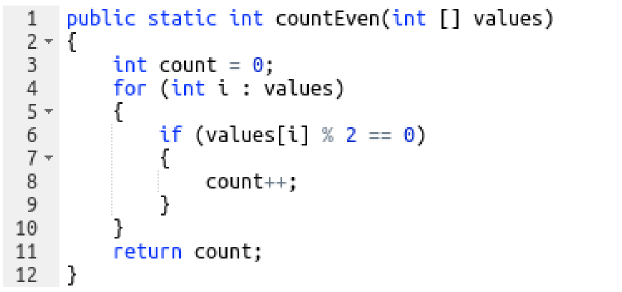
```
 1  public static int countEven(int [] values)
 2  {
 3      int count;
 4      count = 0;
 5      for (int i = 0; i < values.length; i++)
 6      {
 7          if (values[i] % 2 != 1)
 8          {
 9              count = count + 1;
10          }
11          else
12          {
13              count = count;
14          }
15      }
16      return count;
17  }
```

Oplossing

De te nemen refactor-stappen zijn als volgt:

1. Regel 3 en 4 samenvoegen: declaratie en instantiatie van de count variabele kan in 1 regel
2. De for-loop op regel 5 kan vervangen worden door een for-each loop
3. De conditie op regel 7 kan netter: values[i] % 2 == 0
4. Het ophogen van count kan korter: count ++
5. Het else statement kan weg: self-assignment voegt niks toe

De foute inzending van de student

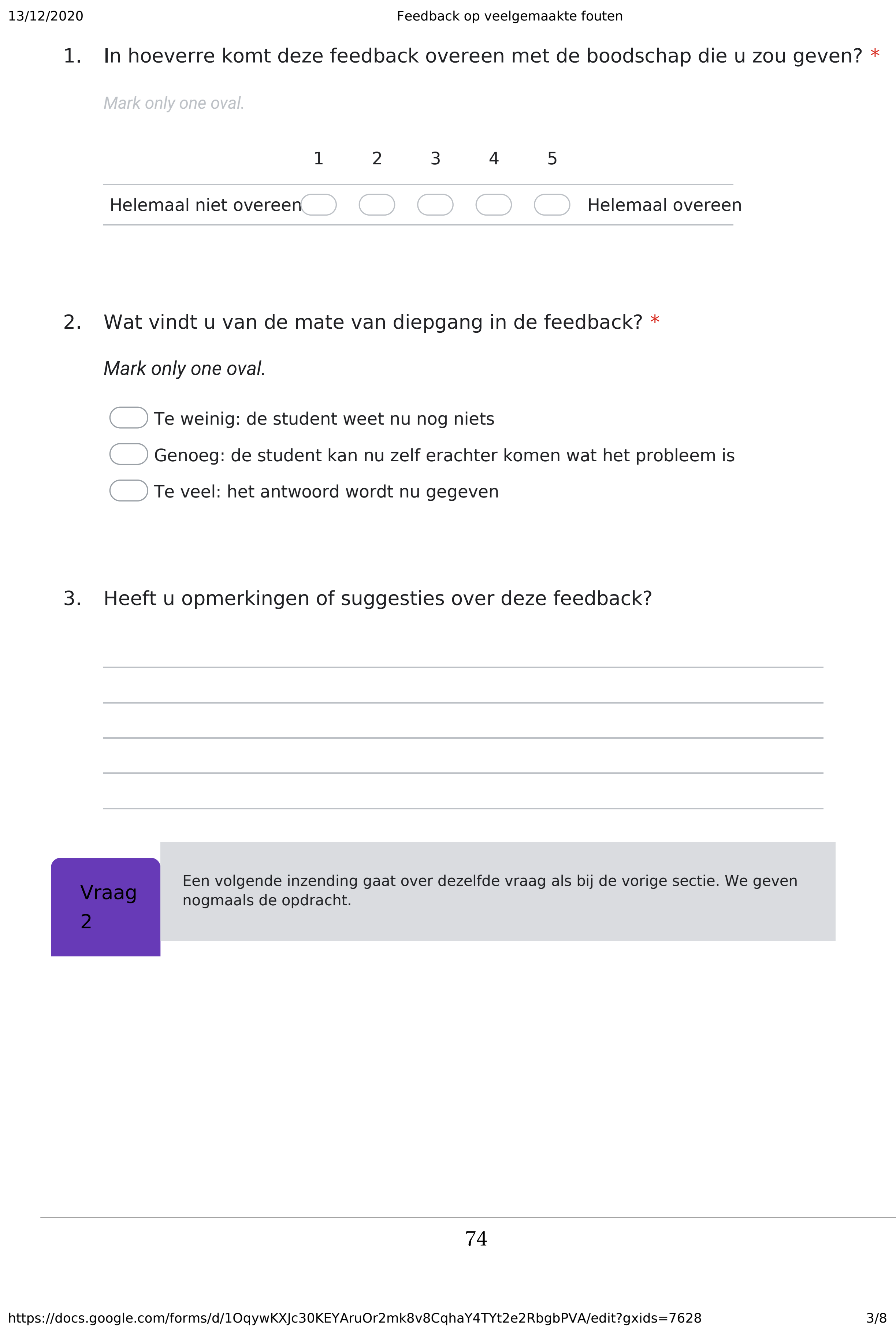


```
 1  public static int countEven(int [] values)
 2  {
 3      int count = 0;
 4      for (int i : values)
 5      {
 6          if (values[i] % 2 == 0)
 7          {
 8              count++;
 9          }
10      }
11      return count;
12  }
```

Omschrijving probleem en feedback

Probleem: Alle stappen behalve stap 2 zijn correct doorgevoerd. Hoewel het type loop goed is gewijzigd, is de betekenis van variabele i veranderd: In plaats van een index, bevat het nu de waarde uit de array op dat moment in de iteratie aan.

De feedback die wordt gegeven aan de student is als volgt:
The return value isn't correct. You changed the for loop to a for-each. This also changed the meaning of the loop variable, from an index to the actual value in the array. Did you forget to change how to access the value from the array?



13/12/2020 Feedback op veelgemaakte fouten

1. In hoeverre komt deze feedback overeen met de boodschap die u zou geven? *

Mark only one oval.

	1	2	3	4	5	
Helemaal niet overeen	○	○	○	○	○	Helemaal overeen

2. Wat vindt u van de mate van diepgang in de feedback? *

Mark only one oval.

○ Te weinig: de student weet nu nog niets

○ Genoeg: de student kan nu zelf erachter komen wat het probleem is

○ Te veel: het antwoord wordt nu gegeven

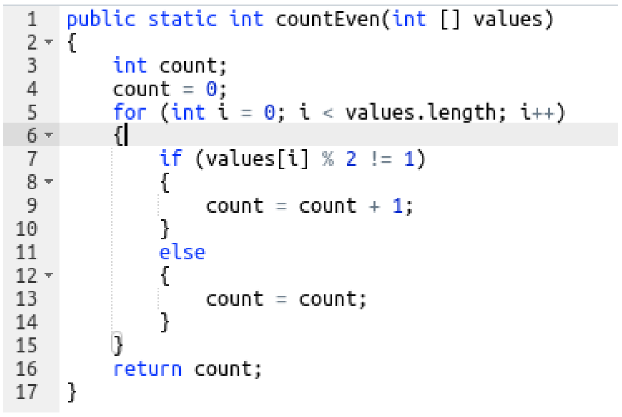
3. Heeft u opmerkingen of suggesties over deze feedback?

Vraag 2

Een volgende inzending gaat over dezelfde vraag als bij de vorige sectie. We geven nogmaals de opdracht.

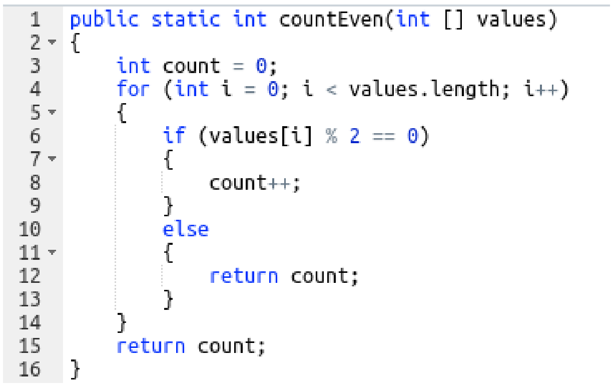
https://docs.google.com/forms/d/1OqywKXJc30KEYAruOr2mk8v8CqhaY4TYt2e2RbgbPVA/edit?gxids=7628 3/8

The countEven method returns the number of even integers in the values-array. Example test case: {1,2,3,4,5} returns 2. You don't have to deal with negative numbers. The solution is already correct, but can you improve this program?



```java
public static int countEven(int [] values)
{
    int count;
    count = 0;
    for (int i = 0; i < values.length; i++)
    {
        if (values[i] % 2 != 1)
        {
            count = count + 1;
        }
        else
        {
            count = count;
        }
    }
    return count;
}
```

De foute inzending van de student

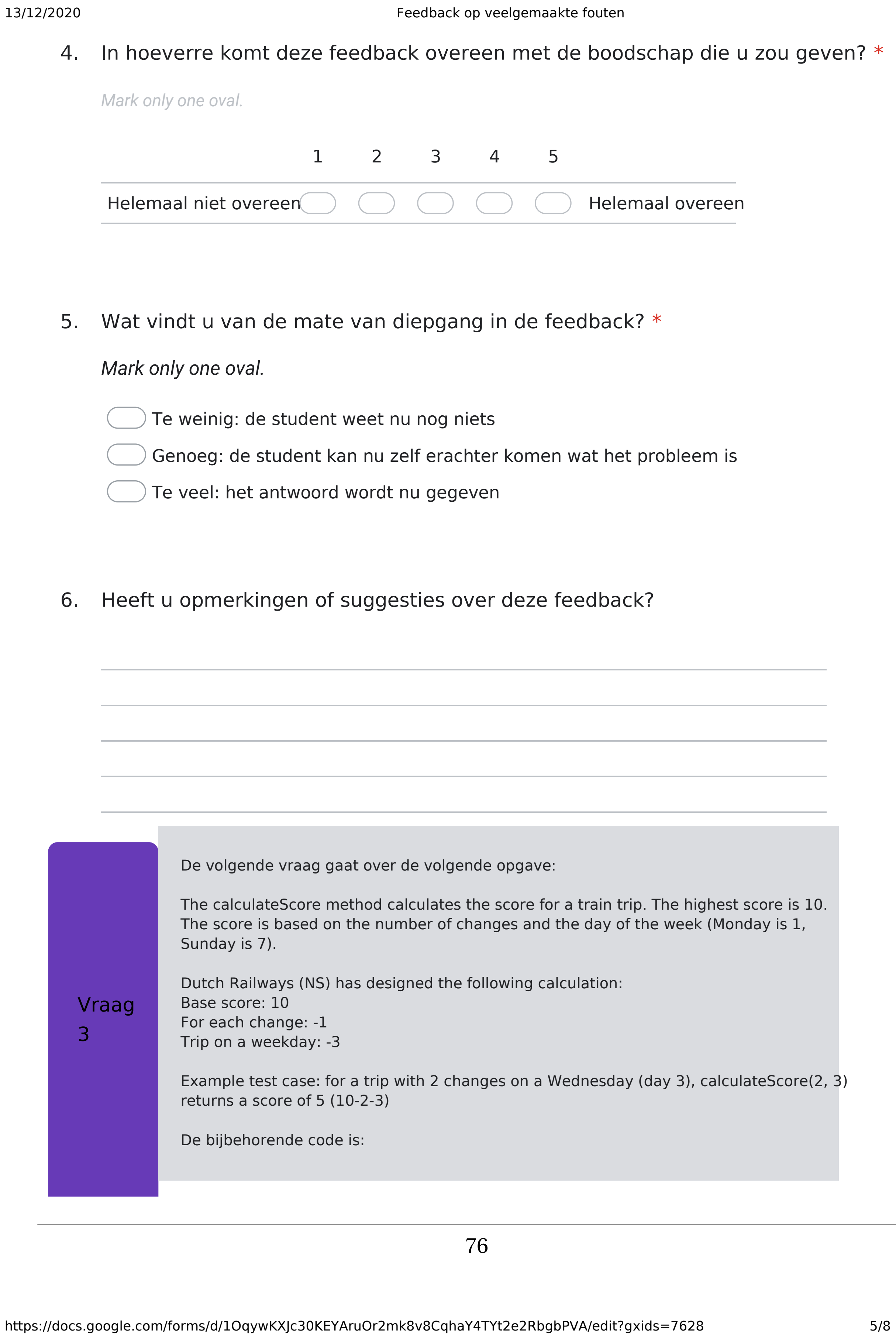


```java
 1  public static int countEven(int [] values)
 2  {
 3      int count = 0;
 4      for (int i = 0; i < values.length; i++)
 5      {
 6          if (values[i] % 2 == 0)
 7          {
 8              count++;
 9          }
10          else
11          {
12              return count;
13          }
14      }
15      return count;
16  }
```

Omschrijving probleem en feedback

Probleem: Bij de volgende inzending zijn alle refactor-stappen doorlopen, maar de methode wordt na het vinden van een oneven getal afgebroken: op regel 12 staat return count;.

De feedback die wordt gegeven aan de student is als volgt:
The return value isn't correct. It looks like not all values are considered. Did you return too early?



13/12/2020 Feedback op veelgemaakte fouten

4. In hoeverre komt deze feedback overeen met de boodschap die u zou geven? *

Mark only one oval.

	1	2	3	4	5	
Helemaal niet overeen	◯	◯	◯	◯	◯	Helemaal overeen

5. Wat vindt u van de mate van diepgang in de feedback? *

Mark only one oval.

◯ Te weinig: de student weet nu nog niets

◯ Genoeg: de student kan nu zelf erachter komen wat het probleem is

◯ Te veel: het antwoord wordt nu gegeven

6. Heeft u opmerkingen of suggesties over deze feedback?

Vraag 3

De volgende vraag gaat over de volgende opgave:

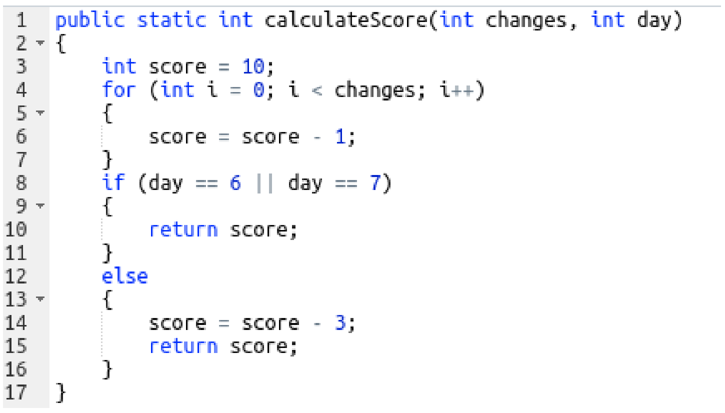
The calculateScore method calculates the score for a train trip. The highest score is 10. The score is based on the number of changes and the day of the week (Monday is 1, Sunday is 7).

Dutch Railways (NS) has designed the following calculation:
Base score: 10
For each change: -1
Trip on a weekday: -3

Example test case: for a trip with 2 changes on a Wednesday (day 3), calculateScore(2, 3) returns a score of 5 (10-2-3)

De bijbehorende code is:

https://docs.google.com/forms/d/1OqywKXJc30KEYAruOr2mk8v8CqhaY4TYt2e2RbgbPVA/edit?gxids=7628 5/8



```
1  public static int calculateScore(int changes, int day)
2  {
3      int score = 10;
4      for (int i = 0; i < changes; i++)
5      {
6          score = score - 1;
7      }
8      if (day == 6 || day == 7)
9      {
10         return score;
11     }
12     else
13     {
14         score = score - 3;
15         return score;
16     }
17 }
```

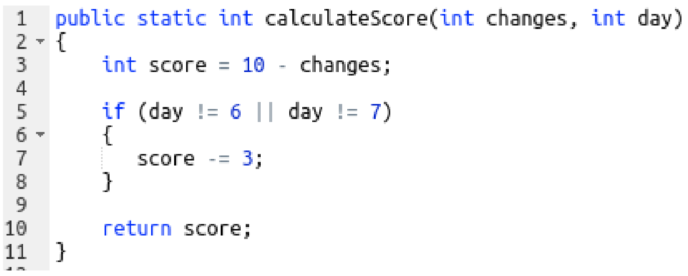
Oplossing

De te zetten refactorstappen zijn:

1. Loop weghalen: dit kan vervangen worden door een berekening: score – changes
2. Initialisatie en berekening samenvoegen: int score = 10 – changes
3. Berekening in regel 14 vervangen door: score -= 3;
4. if-statement herschrijven zodat er maar 1 return nodig is:

```
if (day != 6 && day != 7){
 score -= 3;
}
```

De foute inzending van de student



```
public static int calculateScore(int changes, int day)
{
    int score = 10 - changes;

    if (day != 6 || day != 7)
    {
       score -= 3;
    }

    return score;
}
```

Omschrijving probleem en feedback

Probleem: Alle refactor-stappen zijn goed gedaan, behalve de laatste. In plaats van de and-operator heeft de student een or-operator gebruikt, waardoor er altijd 3 van de score wordt afgehaald.

De feedback die wordt gegeven is de volgende:

The return value isn't correct. Look closely at the used logical operator(s). When do they evaluate to true?

7. In hoeverre komt deze feedback overeen met de boodschap die u zou geven? *

Mark only one oval.

	1	2	3	4	5	
Helemaal niet overeen	◯	◯	◯	◯	◯	Helemaal overeen

8. Wat vindt u van de mate van diepgang in de feedback? *

Mark only one oval.

- ◯ Te weinig: de student weet nu nog niets
- ◯ Genoeg: de student kan nu zelf erachter komen wat het probleem is
- ◯ Te veel: het antwoord wordt nu gegeven

9. Heeft u opmerkingen of suggesties over deze feedback?

Opmerkingen

Heeft u verder nog vragen of opmerkingen?

10. Opmerkingen