MASTER'S THESIS

DIAGNOSING REFACTORING DANGERS: FUNNELING LONGLISTED RISKS INTO A NARROW VERDICT

Brinksma, Wouter

Award date:
2023

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain.
• You may freely distribute the URL identifying the publication in the public portal.

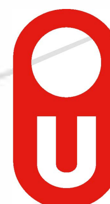
Take down policy
If you believe that this document breaches copyright please contact us at:

pure-support@ou.nl

providing details and we will investigate your claim.

Downloaded from https://research.ou.nl/ on date: 22. Apr. 2025

Open Universiteit
www.ou.nl



Diagnosing Refactoring Dangers: Funneling Longlisted Risks into a Narrow Verdict

by

W.D. Brinksma

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University, faculty of Management, Science and Technology
Master Software Engineering
to be defended publicly on Friday 15 December, 2023 at 15:00.

Student number:		
Course code:	IM9906	
Thesis committee:	dhr. dr. ir. Harrie Passier (chair & supervisor),	Open University
	dhr. prof. dr. Lex Bijlsma (supervisor),	Open University

Open Universiteit
www.ou.nl

ACKNOWLEDGEMENT

All's well that ends well, a phrase that might be exclaimed as a sigh of relief at the end of a complex project. It should therefore be no surprise that it can be said at the end of a master's program, while finishing one's thesis. And considering you are currently reading a master thesis, it might even have occurred in this specific context as well. This I know, and it was.

Looking back, I am proud of the research described in this thesis and the tool that was developed, but I also know that it was not a one-man job. This work would not have been possible without the help of several people whom I would like to thank below.

I want to thank my supervisors Harrie Passier and Lex Bijlsma for their guidance and our scientific discussions. I have enjoyed our collaboration very much and am grateful for the opportunities given to me. I look forward to working together in the future.

I am eternally grateful to my partner Alissa Lohues, and my parents Richard and Grietje Brinksma. In the past months, I was not able to make enough time for them, but they always made more than enough time for me and supported me through the whole process.

Lastly, I would like to thank my Team Leader, Soon Hee Santema, for providing me with the time, space, and opportunity to finish my studies. I want to thank both Soon Hee and Pier Santema for providing me with a quiet place to study and for their pleasant company.

Wouter Brinksma
10 December 2023

Contents

Summary **v**

1 Introduction **1**
1.1 Behavior preservation . . . 1
1.2 Illustrative refactoring examples . . . 2
1.3 General research goal . . . 4
1.4 Reading guide . . . 5

2 Related Work **7**
2.1 Previous work . . . 7
2.2 Academic literature . . . 8

3 Research Method **11**
3.1 Research goal . . . 11
3.2 Knowledge gaps . . . 11
3.3 Research questions, methods and validation . . . 12

4 Refactoring Case **15**
4.1 Case description . . . 15
4.2 Results . . . 18
4.3 Considerations . . . 19

5 Refactoring Diagnosis Model **21**
5.1 The model . . . 21
5.2 Detection automation . . . 23
5.3 Model architecture . . . 26

6 Refactoring Diagnosis Plugin **29**
6.1 Architecture of the plugin . . . 29
6.2 Program graph . . . 30
6.3 Representing possibly nonexistent code locations . . . 31
6.4 Refactoring . . . 32
6.5 Microstep . . . 32
6.6 Detector . . . 33
6.7 Sets, Streams & Generators . . . 34
6.8 Subdetector . . . 35
6.9 Analysis . . . 36
6.10 Verdict . . . 36
6.11 Changing the graph . . . 37
6.12 List of implemented microsteps and detectors . . . 38
6.13 Plugin demonstration . . . 39

7 Validation **43**
7.1 Pull Up Method 43
7.2 Combine Methods into Class 45

8 Discussion **47**
8.1 General topics of interest 47
8.2 Limitations of the study 48

9 Conclusion **49**

10 Future Work **51**

Bibliography **53**

SUMMARY

As the real world changes, software that works in that world has to adapt. Initially, a program is developed with a specific design in mind, but new requirements later introduced to the software do not always match that existing design. Therefore, adaptions have to be made to make the new requirements fit, but these can lead to lower software quality. To improve the quality of the software, the internal structure of the program has to be revised while the behavior of the program stays the same. This activity is called *refactoring*. We pay special attention to how it can be guaranteed that a program exhibits the same behavior after a refactoring has been performed. This is known as behavior preservation.

Behaviour preservation analyses are often incomplete, little or no insight is given into the reasons why a refactoring is rejected, and no solutions are given for each of the reasons for rejection. However, programmers prefer that a refactoring engine does not only reject the application of a refactoring. Instead, they prefer insight into the reasons for rejection and possible solutions to satisfy the failed preconditions. And to teach students to refactor, it is important that they understand the complexities of refactoring and learn how to deal with these complexities. Therefore, it is important that, for example before starting a refactoring task, an overview is given of the dangers that may arise.

Our goal is to design and implement a system for the automatic identification of refactoring dangers that supplies information to generate an advice from. This design should be implemented in a reference implementation (prototype) which can later be extended. Specifically, we focus on how our current knowledge can be implemented and extended, how a modular system can be developed, and how false positive results can be eliminated.

In this thesis, we present a new model to detect refactoring dangers and a first implementation in the form of an Eclipse plugin called *ReFD*. The model detects dangers by splitting a refactoring up into microsteps that have associated potential risks. A detector examines the code context to determine if the potential risk is present in the code, thus becoming an actual risk. The resulting actual risks are evaluated by a verdict mechanism to remove false positives.

The Eclipse plugin has been shown to be able to detect dangers in a refactoring by closely implementing the aforementioned model. It supports two refactorings, *Pull Up Method* and *Combine Methods into Class*, that are based on Fowler's descriptions and the results of a case study that was performed. In this case study, we reviewed the problems students and experts encounter while performing the Pull Up Method refactoring on a code example.

Streams were developed to chain subdetectors to query the codebase. These streams of program locations can be processed by chaining subdetectors as operations on the stream, much like how Java streams can form a chain of operations. Subdetectors query the codebase by querying a graph representation of it. For this, we use an existing tool called Atlas.

Finally, ReFD was validated against the earlier described case study code example. Both implemented refactorings were tested on multiple inputs and yielded the expected results.

1

Introduction

As the real world changes, software that works in that world has to adapt. Initially, a program is developed with a specific design in mind, but new requirements later introduced to the software do not always match that existing design. Therefore, adaptions have to be made to make the new requirements fit, but these can lead to lower software quality. To improve the quality of the software, the internal structure of the program has to be revised while the behavior of the program stays the same. This activity is called *refactoring* [14]. Mens and Trouwé [10] described the refactoring process as a number of separate activities:

1. Identify where the software should be refactored;
2. Determine which refactoring(s) should be applied to the identified places;
3. Guarantee that the applied refactoring preserves behavior;
4. Apply the refactoring;
5. Assess effects on software quality characteristics or development process;
6. Maintain consistency between the refactored code and other software artifacts.

Step three has our special attention: How can it be guaranteed that a program exhibits the same behavior after a refactoring has been performed? This is known as behavior preservation [14].

1.1. Behavior preservation

One way of guaranteeing behavior preservation is to formally *prove* beforehand the program semantics. For some languages with formally defined semantics, one can prove that some refactorings preserve the program semantics [16]. However, for most complex languages like Java, this is extremely difficult, if not unfeasible [18].

Another technique that can be employed is *testing* [10]. With testing, tests specifying the desired behavior determine whether the specified relationship between input and output has been preserved after the refactoring has been performed [4].

However, the use of testing in this manner has several problems. These include:

- Verification takes place after a refactoring has taken place;
- Errors can remain undetected if test cases do not fully cover all the desired behavior;
- Failed test cases do indicate that certain desired behavior is no longer being exhibited, but they do not always clearly indicate where the cause can be found in the code and how they can be remedied;
- Tests can become invalidated by structural changes in the code, which is non-trivial to resolve [15].

An example of the last bullet is the Extract Method [4] refactoring as inverse refactoring of Inline Method [4]. Extract Method has many variants. Depending on which parameters are selected, the original or another program is created. If the result is a different program, the original test may not be applicable because the signature of a method can be different from the original.

Behavior preservation can also be achieved by specifying *refactoring preconditions* in advance that must be met before a refactoring is performed [10, 14]. This technique is applied in many refactoring engines such as Eclipse[1] and NetBeans[2]. Much like testing, several problems are associated with this technique. These include:

- Refactoring engines may have overly weak and overly strong preconditions, allowing incorrect transformations [17] and preventing correct transformations [11];
- The number of ways to establish an enabling condition is often too limited (see explanation after Example 1.2.2);
- It is not always clear why a refactoring is rejected and no remedial action or sufficient alternatives are suggested [1];
- Refactor engines are not open for extension, i.e. it is not possible to add other refactorings.

In summary, behavior preservation analyses are often incomplete, little or no insight is given into the reasons why a refactoring is rejected, and no solutions are given for each of the reasons for rejection. However, programmers prefer that the refactoring engine does not only reject a refactoring application. Instead, they prefer insight into the reasons for rejection and the possible solutions of how to satisfy the failed preconditions. With this insight, they are able to manually fix these problems afterward [20].

1.2. ILLUSTRATIVE REFACTORING EXAMPLES

The following two examples illustrate the problems mentioned in Section 1.1. Example 1.2.1 presents a situation in which different choices can be made to fulfill a refactoring's precondition. Example 1.2.2 presents a situation when a developer should receive a warning of a possible unwanted situation in code. Both examples use Java as the programming language. The end of an example is denoted by the character □.

[1] https://eclipseide.org/
[2] https://netbeans.apache.org/

Example 1.2.1 Consider the source code in Figure 1.1. Now assume that we wish to apply a Move Method [4] refactoring and move method `Source.method` to class `Target`.

```
1  class Source {
2
3     private int local = 15;
4
5     public void method(Target target) {
6        target.doSomething();
7        System.out.println("Executing source method with " + local);
8     }
9  }
10
11 class Target {
12
13    public void doSomething() {
14       System.out.println("Executing target code");
15    }
16 }
```

Figure 1.1: Example code for Move Method refactoring

```
1  class Source {
2
3     private int local;
4
5     public int getLocal() {
6        return local;
7     }
8  }
9
10 class Target {
11
12    public void doSomething() {
13       System.out.println("Executing target code");
14    }
15
16    public void method(Source source) {
17       doSomething();
18       System.out.println("Executing source method with " + source.getLocal());
19    }
20 }
```

Figure 1.2: Applied Move Method on code from Figure 1.1

A problem that is immediately visible is that attribute `local` is not in scope within class `Target`. Several solutions are possible for this situation. For instance, class `Source` could add a getter to get the value of attribute `local`. Attribute `local` could also be moved to class `Target`, or its visibility could be set to package. If we choose the first option, the resulting code would be that of Figure 1.2. There exist many situations where one of the other options would be preferable. When performing a refactoring, the programmer should be aware of the various possibilities and be able to make an informed choice. □

Example 1.2.2 Consider the starting situation of Example 1.2.1 again and assume we want to move method `Source.method` to class `Target` while class `Target` has a subclass `Sub`, already having a method `method` with parameter `source` of type `Source`. The additional class is shown in Figure 1.3.

```
1  class Sub extends Target {
2
3     @Override
4     public void method(Source source) {
5        System.out.println("Using subclass method");
6     }
7  }
```

Figure 1.3: Class addition to code from Figure 1.1

Note that `Sub.method` has the same signature as the new method `Target.method` that appeared after applying the Move Method refactoring as shown in Figure 1.2. Since the programmer performing the refactoring may not be aware of the existence of `Sub`, there could be a situation of unintended overriding of the `Target.method`. Here, the programmer should receive a warning. □

AUTOMATED REFACTORING BEHAVIOR

In Example 1.2.1, automatic refactoring in Eclipse[3] will choose the option to give attribute `local` package scope (and provide a warning that this has occurred), without mentioning the other solutions. The problem with subclass `Sub` from Example 1.2.2 is ignored by Eclipse.

In case `Source.method` overrides a method in a superclass of `Source`, on the other hand, Eclipse refuses to perform the refactoring, and generates a message "The method invocations to `Source.method` cannot be updated, since the original method is used polymorphically". It is unclear why this refusal occurs because the problem can be solved easily by keeping `Source.method` as a shell that redirects all calls to `Target.method`.

1.3. GENERAL RESEARCH GOAL

The problems mentioned in the previous sections are of interest to professional software developers and students of computer science. Refactoring code can have associated dangers, i.e. possible behavioral changes of the code, which are defined in detail in Chapter 5. For professional developers, it is important to understand exactly which dangers are involved before a refactoring can be applied and how each of these dangers can be solved. Depending on this, the decision can be made whether or not to carry out a refactoring and how each of the dangers can be best solved. If software engineers have a detailed insight into what the dangers are, they will be better able to judge on that basis whether a refactoring should be carried out and in what form [20].

To teach students to refactor, it is important that they understand the complexities of refactoring and learn how to deal with these complexities. For this purpose, it is important that, for example before starting a refactoring task, an overview is given of the dangers that may arise. Various didactic approaches are conceivable for using this insight in programming education.

To provide such an overview of refactoring dangers and solutions, a new refactoring diagnosis tool is needed. Specifically, the tool should be able to provide a full overview for each supported refactoring, including its dangers and all possible solutions for these dangers. For each refactoring, the tool should be able to pinpoint and explain dangers based

[3] At least in one version. The behavior of Eclipse tends to change with each new version.

on the code context the refactoring is applied to. These dangers can have multiple solutions, such as applying subsequent refactorings. The tool should suggest these solutions, for instance in a tree-like fashion, and offer textual advice on its implications. Each node in the tree could represent a choice within the refactoring, which would allow users to tailor the refactoring and think about the implications. Ideally, the tool is integrated into existing IDEs. This makes adoption easier and provides an additional tool on top of an already rich environment.

In persuit of this goal, multiple Open Universiteit students have already worked on this topic. A compact description of their work is given in Section 2.1.

USE CASE EXAMPLE

To provide a more accurate idea of the tool we are envisioning, we provide a short use case example based on the refactoring examples from Section 1.2. However, larger and more complex use cases can be envisioned as well.

In this use case, the tool is a plugin within the Eclipse IDE. It provides a system to select a piece of code, on which refactorings can be applied. We can select `Source.method` in the editor and open a menu to select the Move Method refactoring. Next, the user can choose the class the method should be moved to. After selecting a class, the plugin evaluates all changes that should/could be made in order to carry out the refactoring. This results in a screen which shows a number of refactoring steps. Broadly speaking, in the case of Move Method the first step would be to add `method` to `Target` and the second step would be to remove `Source.method`.

Adding `method` to `Target` results in a possible problem: the newly added method is overridden by `Sub.method`. Now, there is a possibility this is intended by the developer, but perhaps a greater possibility that this is not the case. The tool presents this problem to the user and shows multiple options. One option would be to do nothing, but another would be to apply Rename Method to the newly added method, or to apply Rename Method to `Sub.method`. For each problem resulting from each step, such options are displayed. They are all displayed in one full advice containing all choices.

1.4. READING GUIDE

The remainder of this thesis is organized as follows. Chapter 2 will provide an overview of previous work by the Open Universiteit on this subject and relevant published academic literature. Chapter 3 will detail the research goal, knowledge gaps, and the research method used to gain insight into the way the tool described above can be developed. Chapter 4 provides a small case study of a simple Pull Up Method refactoring example which was used to identify problems people encountered while executing the refactoring. Chapter 5 describes the refactoring diagnosis model we developed to diagnose refactoring dangers. Chapter 6 details the refactoring diagnosis tool that was developed as a result of our research. Chapter 7 shows the results the tool provides when tested on the refactoring case study and compares these with expected values. Chapter 8 contains some interesting topics that might be of value to the reader, together with a discussion on the limitations of this study. Finally, Chapter 9 presents the conclusion of the research, and Chapter 10 provides avenues for future work.

2

Related Work

This chapter details work related to our research. Section 2.1 provides a short description of previous work by Open Universiteit students (master and bachelor), and Section 2.2 describes relevant academic literature.

2.1. Previous work

De Beer [1] was the first student to work on this research topic and showed its novelty. He laid a foundation for a system that generates refactoring advice based on code context and refactoring risks. The underlying model contained the notion of a *Code Context Property* (CCP), which is a code construct associated with a specific refactoring advice. A CCP could be detected by a *Code Context Property Detector* (CCPD).

Next, Verduin [21] and Hilberink [7] simultaneously introduced a new important concept to the refactoring diagnosis theory: the *microstep*. A microstep was envisioned as a more manageable step in a refactoring for which specific risks could be defined. Microsteps mainly come in two types: *add* and *remove*. Depending on the author, more types such as *rename* or *change* also appear. Additionally, Verduin introduced a simplified AST representation that was supplied together with a query language to describe hazardous code patterns. Verduin also created a preliminary overview of hazards connected to known microsteps. Hilberink refined the concept of risk with *Risk-based Refactoring*. Risk-based Refactoring includes the concepts of *Potential Risk* and *Actual Risk*. The former was defined as the possibility that a change in code leads to program behavior preservation break, and the latter as when a potential risk is actually encountered in the context of a particular code context.

In his bachelor's thesis, Wernsen [22] researched source code query systems to support the detection of patterns in source code that can lead to behavior preservation break when a refactoring is applied. From his research, he identified Atlas[1] as the system to best match all requirements. A prototype of the query system was developed, as well as an extension of the Atlas query system called *WQL*.

More recently, work has been started to incorporate the findings described above into a general refactoring diagnosis model. The model is described in Chapter 5 and an earlier version of it formed the basis for this thesis. This model streamlines many of the terms

[1] `https://www.ensoftcorp.com/atlas/`

used in the previous works and selects a core set of them to use. The primary elements of this model are the *Refactoring*, *Microstep*, *Detector*, *Subdetector*, and *Verdict Mechanism*. A refactoring can be divided into microsteps, which in turn have potential risks associated with them. A detector detects if a potential risk is present in a code context, making it an actual risk. To do this, it needs some kind of modular query system. A subdetector is simply one part of such a query that can be reused for multiple detectors. Lastly, when dividing a refactoring into microsteps with associated risks, these might sometimes conflict. Such conflicts lead to false positive detection of those risks. To mitigate this, a subsystem will be necessary. This system is called the verdict mechanism and it filters actual risks. The remaining actual risks are identified as dangers.

The current version of the model included in this thesis has been augmented with insights from the research described in this thesis.

2.2. ACADEMIC LITERATURE

Providing guidance during refactoring to learn this skill could be thought of as less necessary when developers make heavy use of automated refactoring tooling. While we think having a good grasp of refactoring mechanics is always necessary, it is interesting to note that there is evidence for developers being more inclined to refactor their code manually [12, 19, 8]. In particular, the paper by Kim et al. [8] highlights a response from a professional developer asking for code understanding tools to aid in manual refactoring.

It may be that such tools are requested because the process of refactoring is not always clear, even from the teaching material. Fowler's [3] famous book provides an example in the form of the mechanics of the Inline Method refactoring. Trailing the list of mechanics, we find a warning that lists several vague complicating factors which, if one encounters them, should be taken as a signal not to execute the refactoring.

Mens and Trouwé [10] provide an extensive overview of refactoring. Their paper contains a useful refactoring example and a list of activities of the refactoring process, which start with identifying the location and type of refactorings to be applied, to go on to guaranteeing behavior preservation. To achieve the latter, they list testing, weakening the notion of behavior preservation and formal proofs. Unfortunately, they do not list a diagnosis of dangers associated with refactorings. However, research that is somewhat close to ours does exist. Below, we provide a small overview.

Soares [17] presents a tool that checks if a refactoring is safe, i.e. if it preserves program behaviour. This tool works by generating test cases that should pass for the initial program and the refactored version. Although we think using testing for this purpose has problems, Soares did find numerous bugs in the refactoring engine used by Eclipse, some of which led to changes in behavior preservation that were hard to spot.

Opdyke [14] defined behavior preservation in terms of receiving unchanged output between the original and refactored versions of the program, while providing both with the same input. He also suggests that *preconditions* can be used to check if the program behavior will be the same after a refactoring.

Kniesel and Koch [9] introduced a formal model to compose conditional program transformations, i.e. refactorings with preconditions. In this model, a conditional transformation is a pair of a condition and transformation. In particular, a condition is a mapping of a program to a logical value (true or false) and multiple can be strung together by using a conjunction or disjunction. The goal of the conditions is to find if the associated refactoring

will be behavior-preserving. Our model, which is compositional as well, focuses on relaying offending program locations instead of only a logical value to gain insight into actual danger advice. In addition, we introduce the concept of a microstep as the fundamental element of a refactoring for which dangers can be found.

Haendler et al. [6] present a tool that gives the user a program and a description with one or more refactorings to be applied to that program. The tool also provides a UML diagram of the program after successful refactoring and another of the program as-is. Unit tests can be run to assess if the program was refactored correctly and some code quality analysis is also executed. While this is a comprehensive tool, it does not analyze the steps in a refactoring like our proposed tool does and also relies on testing like the tool by Soares.

3
Research Method

This chapter starts with a description of the research goal and requirements for the to-be-developed tool. In the following section, we combine this goal with the refactoring diagnosis model from Chapter 5 and arrive at gaps in our current knowledge. The final section will provide research questions to solve these knowledge gaps, together with the methods used to research each question and validate the results.

3.1. Research goal

To create the tool described in Section 1.3, a system to generate refactoring advice will need to be developed. This advice is dependent on the dangers detected in the selected refactoring. In this thesis, the technology of behavioral preservation utilizing static detection of dangers is central. Rather than preconditions forbidding refactorings, we desire insight into actual dangers and advice on possible remedial actions. Therefore, our goal is to design a system for the automatic identification of refactoring dangers which supplies information to generate an advice from. This design should be implemented in a reference implementation (prototype) which can later be extended. This leads us to the following list of requirements:

- Prior to a refactoring, insight is given to all dangers that are present;
- The tool should be integrated into an existing IDE;
- The tool should use the refactoring diagnosis model introduced in Section 2.1 (and detailed in Chapter 5);
- The implementation of the model should be extensible, i.e. other refactorings can be easily added.

3.2. Knowledge gaps

When we combine the research goal in Section 3.1 with the previous research described in Section 2.1, three knowledge gaps appear. The first gap is the way the Atlas query system can be combined with the previously developed refactoring diagnosis model. As explained in Section 2.1, a tool has been built incorporating Atlas, but this tool does not implement

the refactoring diagnosis model. A reference implementation of the model should be built using the Atlas system to show if refactoring dangers are sufficiently detected by it.

Second, it becomes clear that risks associated with a microstep can be found by detectors, which are made up of subdetectors. As described in Section 2.1, microsteps are a combination of a relatively simple operation such as adding or removing something, and a syntax construct to apply this operation on. This makes them easy to fathom. In contrast, for detectors and subdetectors it is more difficult to see the basic elements they can represent. It is unclear how many there are, how they relate to microsteps, and what the amount of possible reuse of a detector is.

The third and final gap is the verdict mechanism, because the mechanism itself is not well understood. However, we do think it is most likely that each refactoring will have its own verdict function associated with it.

3.3. RESEARCH QUESTIONS, METHODS AND VALIDATION

Following the research goal and the knowledge gaps from the previous sections, the following research question is defined:

How can a refactoring diagnosis system be implemented?

To answer this question, the system will have to be implemented and the refactoring diagnosis model deepened. In general, the primary method of research will be *Design and Creation* [13] combined with *Model Driven Design* [2]. The tool created by Wernsen [22] will serve as a basis for the research and will be completely reworked by implementing the model. This means that in addition to the knowledge gained by extending it, the tool will be able to serve as a basis for future research and development. This leads to the following subquestions, each with the research method and validation specified.

SQ1 *How can a reference implementation of the current refactoring diagnosis model be developed?*

RESEARCH METHOD

It is not yet fully known if the model described in Section 2.1 works as expected. This is because the model has not yet been implemented fully and only partial implementations exist. To explore the way the model can be implemented and to test if it works, we will create a reference implementation with one refactoring: Pull Up Method [4]. This implementation will serve as the basis for further research.

VALIDATION

To validate if the detection of refactoring dangers works, a simple case study for the Pull Up Method refactoring will be performed by selecting an example case and fully reviewing it beforehand. The review will be carried out by experts and students (the target audience). This review will yield risks and dangers that arise from the refactoring. This information can be compared with the information generated by the

reference implementation. The system should provide that same information to the user as well.

SQ2 *How can a modular system of detectors and subdetectors that facilitates reuse and a uniform flow of information be designed?*

Research method

The current model allows for a refactoring diagnosis system to be built in a modular way. We want to further the development of the tool and model by expanding the library of refactorings. This will give us more insight into the way the refactorings can be split up and will help us better list the all basic components of refactoring diagnosis. In addition, we want to improve the flow of information. To this end, the Combine Methods into Class [4] refactoring will be implemented, as it is a larger and more complicated refactoring.

Validation

It can be difficult to correctly divide larger refactorings into a set of detectors that may overlap with other sets of detectors. To validate if the found sets of detectors are correct, the case study of SQ1 will be reused in this context on two combinations of methods. These combinations will be reviewed to determine their dangers beforehand. This information will be compared with the information generated by the tool.

SQ3 *How can the information flowing from detectors be consolidated to a verdict that eliminates false positive dangers?*

Research method

It is currently not understood how we can reliably and uniformly come to a list of dangers from the output of multiple detectors. To understand this, the verdict mechanism described in Section 2.1 will be developed for the refactorings resulting from the previous research questions. This will provide experience with the verdict mechanism and could perhaps be generalized to work for all or most refactorings.

Validation

To validate the results of the verdict function, the previous two refactorings will be taken as test cases. The first refactoring is simple, which should show that the verdict works for simple refactorings. The other refactoring is more complex, which should show that the verdict mechanism can handle such refactorings as well.

4

REFACTORING CASE

This chapter details the case study for the Pull Up Method refactoring. This case study consists of a code example and the description of several Pull Up Method refactorings to be executed upon that code. The case study was sent to a group of six participants (two HBO-ICT students and four experts) to gain insight into the dangers present in the refactoring. Section 4.1 explains the example code studied and a description of the case as it was sent to the group. Section 4.2 provides an overview of the results provided. Section 4.3 covers some general considerations coming forth from the case study exercise.

4.1. CASE DESCRIPTION

The case study begins by detailing the Pull Up Method refactoring, as this specific refactoring will be studied. As supplemental material, a full description of the refactoring is also provided. The case subject is a small piece of code that is part of a larger fictitious employee management system. At some point, the requirements of this program changed - a common fate for software. In this case, the original software contained a class `Employee`, but its logic became outdated. However, some parts of the original software were dependent on the older logic present in `Employee`, which resulted in a difficult situation. To mitigate this, the developers chose to rename `Employee` to `LegacyEmployee` and create a new `Employee` class which extends `LegacyEmployee`. `Employee` overrides some methods to provide the new functionality.

The goal of the case study is to integrate `Employee` into `LegacyEmployee` to arrive at one class that details an employee, instead of two. After this process, `LegacyEmployee` can be renamed back to `Employee`, but this is outside the scope of the case study. Multiple steps are involved in this case study, and participants were asked to describe each step they took and also describe if program behavior changed and in what way.

The example code consists of three files. Figure 4.1 contains a small class with a main function to show a possible way of using the employee classes. Figure 4.3 shows the implementation of `LegacyEmployee`. Lastly, Figure 4.2 shows the implementation of `Employee` and this class contains numbered comments. Each numbered comment signifies a method that should be refactored by applying the Pull Up Method refactoring. The explanation continues after the code examples.

```
1  public class App {
2     public static void main(String[] args) throws Exception {
3        LegacyEmployee emp1 = new LegacyEmployee("Peter", 2999);
4        emp1.salaryBonus(100);
5        System.out.println(emp1);
6     }
7  }
```

Figure 4.1: App.java

```
1  public class Employee extends LegacyEmployee {
2     private String workplace;
3
4     public Employee(String name, double monthlySalary, String workplace) {
5        super(name, monthlySalary);
6        this.workplace = workplace;
7     }
8
9     //#1
10    public void salaryBonus(int bonus) {
11       this.setMonthlySalary((this.getMonthlySalary() + bonus) * 1.01);
12    }
13
14    //#2
15    @Override
16    public String toString() {
17       return this.name + ", earns " + this.monthlySalary + ", works at " + this.
            getWorkplace();
18    }
19
20    //#3
21    public void setName(String name) {
22       this.name = name;
23    }
24
25    //#4
26    public String getName() {
27       return this.name;
28    }
29
30    //#5
31    public void setMonthlySalary(double monthlySalary) {
32       this.monthlySalary = monthlySalary;
33    }
34
35    //#6
36    public double getMonthlySalary() {
37       return this.monthlySalary;
38    }
39
40    //#7
41    public void setWorkplace(String workplace) {
42       this.workplace = workplace;
43    }
44
45    //#8
46    public String getWorkplace() {
47       return this.workplace;
48    }
49 }
```

Figure 4.2: Employee.java

```java
public class LegacyEmployee {
    protected String name;
    protected double monthlySalary;

    public LegacyEmployee(String name, double monthlySalary) {
        this.name = name;
        this.monthlySalary = monthlySalary;
    }

    public void salaryBonus(double bonus) {
        this.setMonthlySalary(this.getMonthlySalary() + bonus);
    }

    @Override
    public String toString() {
        return this.name + ", earns " + this.monthlySalary;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }

    public void setMonthlySalary(double monthlySalary) {
        this.monthlySalary = monthlySalary;
    }

    public double getMonthlySalary() {
        return this.monthlySalary;
    }
}
```

Figure 4.3: LegacyEmployee.java

Some interesting parts of the code should be noted. First, not every part of the code will give as many problems as other parts. For instance, when we look at `Employee.getName()`, we see that the body is exactly the same as that of `LegacyEmployee.getName()`. However, semantic problems do occur, for instance when looking at `Employee.toString()`. This method has a different body than `LegacyEmployee.toString()`. Here, an implementation choice needs to be made.

Second, methods perhaps do not always need to be overwritten. This might be the case for `Employee.salaryBonus(int)` and `LegacyEmployee.salaryBonus(double)`. Note that each version is implemented differently, and more importantly, has one parameter of a differing type. If `Employee.salaryBonus(int)` would be pulled up to `LegacyEmployee.salaryBonus(int)`, this could cause calls to `LegacyEmployee.salaryBonus(double)` to now call `LegacyEmployee.salaryBonus(int)` when the call was provided with an argument of type `int` instead of `double`. Previously, this integer would be automatically converted to a double and `LegacyEmployee.salaryBonus(double)` would be called, but when an overloaded method matches the given argument's type exactly, that method is called instead.

Finally, some methods such as `Employee.getWorkplace()` use local elements of the class. `Employee.getWorkplace()` uses the local field `workplace`, and when that method is pulled up, `workplace` should perhaps come with. This means that some refactorings can cause other refactorings to be necessary.

In this case study, we focus primarily on problems like the automatic parameter conversion described above, and less on semantic issues specific to the refactored program.

4.2. RESULTS

The results gathered from this case study were somewhat free-form. This resulted in personalized differences between results. Because of this, we review the results in a qualitative manner.

PARTICIPANT #1 (STUDENT)

This participant started by noting that methods in `Employee` labeled with comments #3, #4, #5, and #6 are identical to their counterparts in `LegacyEmployee`. This meant they could be removed from `Employee`.

Next, methods #7 and #8 dealing with the workplace were identified and the participant correctly identified the field `workplace` as connected to these. If these methods should be pulled up, and so should `workplace`.

Finally, methods #1 and #2 are refactored. The participant identified that the implementations in `Employee` and `LegacyEmployee` differ. Their chosen solution was to add a private boolean value called `legacy` to `LegacyEmployee`. The constructor in `Employee` was pulled up to `LegacyEmployee`. This instructor sets `legacy` to false, the original constructor sets it to true. Both methods #1 and #2 now also make use of `legacy` and depending on its value, either logic originally associated with `Employee` or `LegacyEmployee` is run.

PARTICIPANT #2 (STUDENT)

This participant started with method #1. They provide a detailed account of the problems involved. Specifically, they note that implicit conversion of doubles to integers is disallowed, so if the result of the refactoring would only be `LegacyEmployee.salaryBonus(int)`, this would induce implicit conversion errors. Because of this, the participant chose not to move method #1 and instead changed its body to refer execution by calling `Employee.salaryBonus(double)`, thus not following the case description. The semantic differences between both methods were noted, and a way was found to factor out the magic number present in `Employee.salaryBonus(double)`, reducing this logic to one method.

Next, method #2 is refactored in a similar way as participant #1 did, this time checking if a workplace exists within the object or not. This is done by a specialized method, which always returns false in `LegacyEmployee`. In `LegacyEmployee`, this method is also introduced, but in this case it checks if `workplace` is null or empty. Methods #3, #4, #5, and #6 were identified to be identical in both classes, which means they could be safely discarded from `Employee`.

Lastly, methods #7 and #8 are refactored. The participant identifies that the `workplace` field should be pulled up along with the methods, as well as the new method introduced when refactoring method #2.

PARTICIPANT #3 (EXPERT)

This participant did almost the same as participant #2, with one exception. When applying Pull Up Method to method #2, instead of also refactoring field `workplace` from `Employee`, they chose to simply create a copy of `workplace` in `LegacyEmployee`, which is null by default.

PARTICIPANT #4 (EXPERT)

This participant also did almost the same as participant #2, but in this case when refactoring method #2, `workplace` was also pulled up and made protected instead of its original `private` access modifier. This meant that methods #7 and #8 did not have to be pulled up as well.

PARTICIPANT #5 (EXPERT)

When refactoring method #1, this participant created `LegacyEmployee.salaryBonus(int)` which overloads `LegacyEmployee.salaryBonus(double)`. They also correctly identified the problem of automatic parameter type conversion that could occur, as explained in Section 4.1. To combat this problem, they choose to change the name of `LegacyEmployee.salaryBonus(int)` to `LegacyEmployee.salaryBonusNew(int)`.

Next, the participant noted that refactoring method #2 results in a naming conflict because class `LegacyEmployee` would contain two methods `toString()`. They chose the same solution participant #4 chose.

The rest of the refactoring process is identical to that of participant #2.

PARTICIPANT #6 (EXPERT)

This participant begins by noting that test cases should first be constructed. Next, they identified the same problems with method #1 as participant #5 did, and they chose the same solution.

The rest of the refactorings are carried out in the same way as participant #2, with the exception of method #2, which results in one implementation that uses `workplace`. The class `LegacyEmployee` now contains two constructors, though its original constructor now fills `workplace` with the value `"Unknown"`.

4.3. CONSIDERATIONS

All participants note that some methods are the same and that when this is the case, no problems occur when applying the Pull Up Method refactoring. Only two participants chose to overload the `salaryBonus(...)` method. In doing so, they correctly identified the automatic type conversion problem and acted accordingly by renaming the method. One additional participant identified the type conversion problems. However, most participants always tried to harmonize both implementations into one, which had to be done for the `toString()` method. The way this was carried out, broadly speaking, was by introducing some element to `LegacyEmployee` (or reusing an existing element) to determine if logic original to `Employee` or `LegacyEmployee` needed to be run.

We can generalize these findings to problems that can be identified when adding or removing a method. These potential problems are listed below.

- When removing a method, the method could still be called. This is not a problem if a superclass also defines a method with the same signature;
- When adding a method to a class, a method with the same signature could be already present.
- When moving a method, a local field or method might be used, thus potentially breaking the connection to it.

5

Refactoring Diagnosis Model

To be able to identify and reason about the dangers of refactoring, we need a model of refactoring that allows us to. This chapter first introduces this model in the form of a Ubiquitous Language (UL), after which we will explain how dangers can be automatically detected. The model presented here differs from the model that was available before this research project started because new insights have been added to it. We provide an example for all elements of the model.

5.1. The model

The core of our refactoring diagnosis model is that behavior preservation can only be broken if the code of an application is modified. Therefore, we have to analyze the steps of a refactoring for actions that change the code. We assume that the refactoring danger analysis takes place once prior to a refactoring: the user is shown all dangers prior to the refactoring and possible solutions to prevent them.

Step

Fowler [4] defines a refactoring as a change made to the internal structure of software, maintaining its observable behavior. A refactoring can be divided into a sequence of smaller behavior-preserving transformations called steps. After executing a step, the software should be able to be compiled and run.

Example 5.1.1 Suppose we move a method `m` from class `A` to `B`. The first step is to examine all program elements used by `m`, after which we copy `m` to class `B`, naming it `m'`, and adjust it so it can still access the same elements as `m`. In the second step, we either change `m` to call `m'` or replace every call to `m` with a call to `m'`. □

If we break up the steps shown in Example 5.1.1, we find that they consist of two smaller entities: the removal of a syntax construct and the addition of one. While a step is behavior preserving, its components are not. We call such components microsteps.

Microstep

Code is modified when a syntax construct is added or removed. Renaming a syntax element, such as the name of a class or method, can be considered as first removing the element followed by adding a new element. The number of syntax constructs on which both

modifications are applicable is made up of all the syntax elements of the programming language. This number is limited too. The combination of a modification (add, remove) and a syntax construct (class, method, attribute, ..., etc.) is what we call a microstep. Adding or removing a syntax construct in a program can have significant impacts on its behavior and functionality.

Example 5.1.2 If class `C` contains an override for inherited method `m` and we remove `m` without thinking about `m` still being present in a superclass of `C`, calling method `m` on an object of type `C` is still possible, but this will execute the inherited method instead of the local one, thus possibly changing the program behavior. □

Because, as seen in Example 5.1.2, a microstep can lead to changes in program behavior, a microstep is not behavior-preserving and carries one or more specific potential risks.

Potential risk

Any microstep carries some risks, i.e. it *can* lead to changes in program behavior.

Example 5.1.3 When we want to add a method `m` to a class `C`, a matter of overriding could unintentionally happen. □

A potential risk is a problem of a microstep that in isolation, i.e. without considering subsequent microsteps and the program code the refactoring is applied to, can change the behavior of a program. For each microstep, all potential risks can theoretically be summed up in advance. It depends on the code context whether a potential risk is an actual risk.

Actual risk

By examining the code context on which the refactoring is applied, it becomes clear if a potential risk is actually present in the code, i.e. one that possibly leads to changed program behavior after the refactoring is completed.

Example 5.1.4 Following Example 5.1.3, when we want to add a method `m` to a class `C`, a matter of overriding happens when class `C` has one or more superclasses and one or more of these superclasses owns a method with the same signature, but different semantics, as method `m`. The same type of risk happens in case one or more subclasses of class `C` own a method with the same signature, but different semantics, as method `m`. If class `C` has no superclass or subclass, there is no risk of overriding. □

An actual risk is a problem of a microstep in the program code the refactoring is applied to, that in isolation, i.e. without considering subsequent microsteps, changes the behavior of the program. Depending on the other microsteps, an actual risk can turn into a danger.

Danger

An actual risk does not always change program behavior eventually. This depends on the complete sequence of microsteps executed as the implementation of the refactoring.

Example 5.1.5 Classes `C` and `D` both have a method `m`. If we move `m` from `D` to `C` as part of a Move method refactoring, an actual risk of double definition takes place and could turn into a danger. However, when a subsequent microstep removes or renames one of the `m` present in `C`, the danger is not present any longer. □

A danger is defined as an actual risk, caused by a microstep, which is not mitigated by subsequent microsteps. This does not mean that the behavior of the program as it stands will necessarily change. If in Example 1.2.2 method `target.method(source)` is only called in situations where the dynamic type of `target` is `Target` rather than `Sub`, no observable change results. In this case, the danger can not be detected by testing. Static analysis, however, can detect the danger. If the behavior does change, this might lead to an error during runtime.

ERROR

Improper refactoring can lead to errors being introduced into a codebase. There are two kinds of errors: run- and compile-time.

Example 5.1.6 When the override method `m` is removed from class `C` in Example 5.1.2, the method might have had some specific effect (i.e. an internal list always being ordered a specific way) needed for objects of type `C` to operate correctly. Now with method `m` removed, `C` will fall back on the superclass implementation of `m` which does not have the desired effects. If other parts of `C` expect these effects to be present, this might result in runtime errors. ☐

Figure 5.1 shows all types of risks we distinguish in relation to the types of contexts mentioned. The last stage shows runtime errors resulting from dangers.

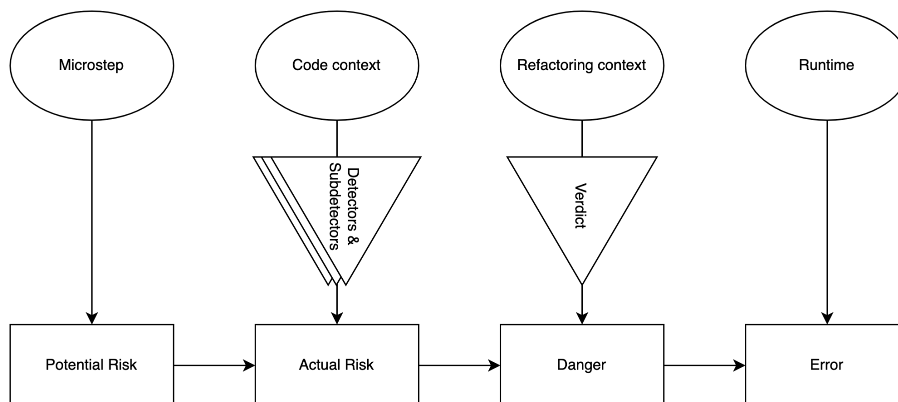


Figure 5.1: Type of risks

Intermediate stages of a refactoring can lead to compile-time errors as well.

Example 5.1.7 When as part of the Move method refactoring a method has been moved to another class and old references to the method are not yet updated to the new method's location, an error results during compilation.

Both types of errors are out of scope in this article. It should be noted that compilation errors can show up as dangers as well.

5.2. DETECTION AUTOMATION

To be able to automatically detect dangers in a refactoring, we developed a conceptual system consisting of detectors, subdetectors, and a verdict mechanism. Below, each of them will be detailed and their interconnections will be explained.

DETECTORS

To be able to determine automatically whether a potential risk is an actual risk, we need a function that inspects the code context. We call such a function a detector.

Example 5.2.1 In case of checking if class `C`'s superclasses contain a specific method `m`, an informal version of the function looks like Figure 5.2 below.

```
1  IF C is a class
2    AND C has superclasses
3    AND m exists in one of these superclasses
4  THEN list all program locations needed for further analysis
```

Figure 5.2: Informal detector version

□

Example 5.2.1 illustrates a detector. A potential risk can be described using an if-then rule. The if-part sums up the combination of properties that should be present in the code context. The then-part pinpoints all the information needed for further processing. It is very useful if a detector does not produce a boolean result but rather a set of program locations where the actual risk occurs. That will enable the programmer to inspect these locations to determine whether the risk leads to behavioral changes, and if so, to perform some remedial change.

Analyzing the potential risks associated with a particular microstep will, in general, take more than one detector. For instance, if we remove a method, we need a detector that will produce all program locations where that method is called. But we will also need a second detector, such as Example 5.2.1 illustrates, that will point to any superclass method that the method to be removed is overriding, because calls to that superclass method will no longer through dynamic binding activate the removed method, thereby possibly changing program behavior.

Detectors are not specific to a particular microstep. For instance, the detector that searches for superclass methods being overridden is also useful when a new method is added: it may be that the new method's name inadvertently leads to it overriding an existing superclass method, again possibly changing program behavior. Hence with every microstep, we associate a number of detectors that look for actual risks, and every one of these detectors is associated with a number of microsteps – a many-to-many relation.

Each boolean sub-expression in the if-part of a detector can be translated into a function querying the code context on the existence of a property. These functions are called subdetectors.

SUBDETECTORS

We have defined a detector as a software unit that will pinpoint program locations with actual risks. But these may be thought of as composed of even simpler units, called subdetectors.

Example 5.2.2 An excerpt of Example 5.2.1 which shows two examples of subdetectors is shown in Figure 5.3 below.

```
1  AND C has superclasses
2  AND m exists in one of these superclasses
```

Figure 5.3: Informal subdetector examples

□

As shown in Examples 5.2.1 and 5.2.2, the detector that searches for superclass methods being overridden may be viewed as the composition of two subdetectors: one that, given a class, produces its ancestors in the inheritance hierarchy; and one that, given a class and a method, produces any occurrence of a method with the same name and signature in that class. We can generalize this by saying that a subdetector always operates on a program location and results in a set of program locations. This also makes it possible that the subdetectors can be successively chained together to form a similar expression such as shown in Example 5.2.2. A subdetector is not a detector, because a subdetector does not determine whether a potential risk is an actual risk.

Again there is a many-to-many relationship: subdetectors are not specific to a particular detector, and a detector will normally use several subdetectors. Note that, different from our concept of detector, subdetectors tend to be chained.

To be able to determine for a particular refactoring whether an actual risk is a danger we need a function that analyzes all the actual risks in the refactoring context, i.e. the code context and the set of steps of the refactoring. We call this analysis function a verdict function.

VERDICT MECHANISM

Producing appropriate advice for an entire refactoring, or even a step, necessitates more than just producing the actual risks found by the detectors. It may be that an actual risk associated with one microstep is mitigated by a subsequent one.

Example 5.2.3 If we perform a change in some method by first adding the new version and then removing the old one, one of the detectors could report an actual risk of a double method definition because at one point during the change, the same method declaration appears twice. However, the actual risk of a double definition does not cause problems because it disappears when the old version is removed. The same holds if we first remove the old version and then add the new one with the actual risk of a missing method definition.
□

Superfluous warnings, such as in Example 5.2.3, had better be removed from the advice; this necessitates a 'verdict' at a level higher than that of microsteps, at the refactor level. The verdict mechanism is a refactoring-specific operation on program location sets resulting from detectors, which has three options: results from detectors can either be fully accepted, partially accepted, or not accepted at all. Whereas detectors and subdetectors can be reused over and over again, verdicts so far have not been decomposed, and one must be custom-built for every refactoring.

5.3. MODEL ARCHITECTURE

A formal difference between subdetectors and detectors is that subdetectors do not answer a question about the presence of an actual risk. But as we have determined that a detector can answer this indirectly through the set of code locations produced, subdetectors and detectors have the same inputs and outputs, all returning just a set of code locations. It is, however, practical to keep the libraries for detectors and subdetectors separate, as these are used for different activities: respectively, analyzing a microstep by calling the associated detector for each of its potential risks, and implementing a detector.

As microsteps are only studied as phases in the execution of a refactoring, detectors are only called for a given microstep to assess which of its inherent potential dangers are actual dangers in the context of the given program, and subdetectors are only called by detectors, these libraries exhibit a *strictly* layered architecture. However, as all communication between these objects proceeds through sets of code locations, the simplified syntax tree representation of the program to be refactored should be globally accessible.

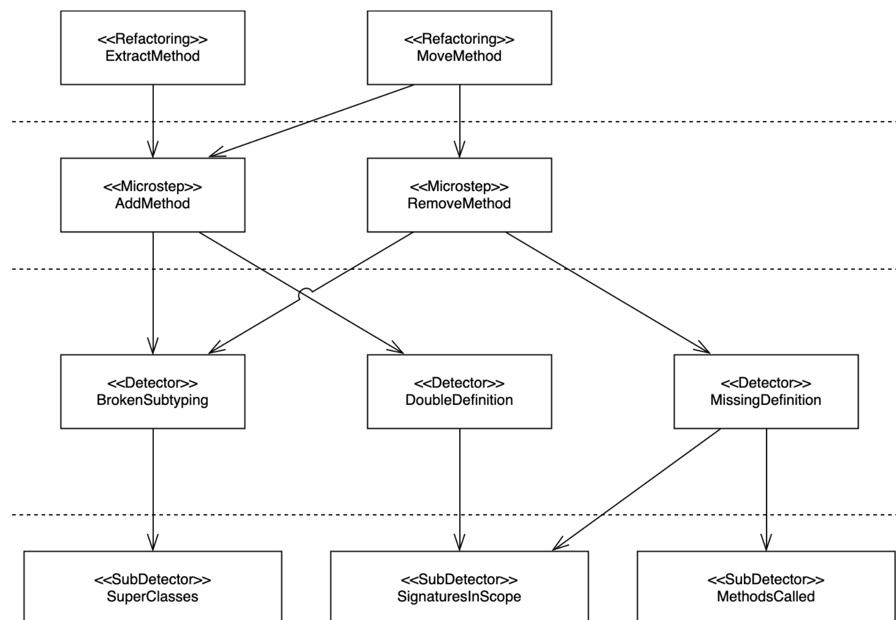


Figure 5.4: Layered architecture

Figure 5.4 shows how the core concepts of the model can be divided into distinct layers. The elements have been tagged with a stereotype, e.g. `<<Detector>>`, to point out their role. However, there are still various possibilities for their implementation. For instance, the elements may be realized as classes, and then the stereotypes will be an interface these classes implement. However, as these might be attributeless singleton classes containing just one static function, an implementation as pure functions may be regarded more natural. In that case, the stereotypes may become their common function type or the container in which the functions are kept. Which is more feasible depends on the language chosen for the tool.

To present the model and its constituent relationships in a visual form, Figure 5.5 shows the domain model of the concepts discussed. An implementation of this model is presented in Chapter 6.

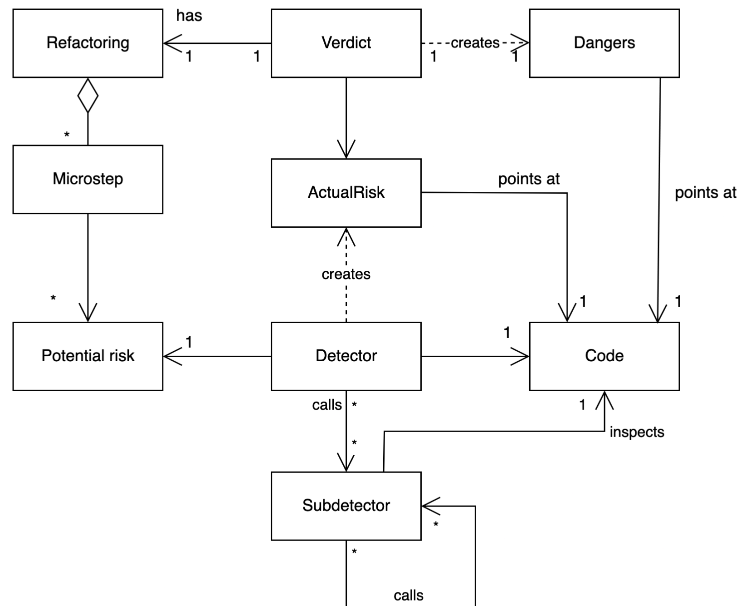


Figure 5.5: Domain model of the concepts

6

Refactoring Diagnosis Plugin

This chapter will present the tool that was developed to better understand the research domain and answer the research questions. The tool takes the form of an Eclipse Plugin[1] named *ReFD*[2]. To explain how the plugin works, we consider a simplified version of the implemented Pull Up Method refactoring and analyze its refactoring dangers. We will highlight each important step in the analysis, which will explain the integration and operation of the different elements of the system.

First, we will explain the architecture of the plugin, after which all elements will be explained in turn.

6.1. Architecture of the plugin

The architecture of the plugin is based on the model architecture described in Section 5.3. The model architecture is a layered architecture, and we have chosen to use the same kind of architecture for the plugin. Figure 6.1 shows this architecture.

The architecture is a layered architecture with several special layers that may be used by all layers. Generally speaking, the architecture consists of two main elements: the plugin-specific logic and the domain model. First, the plugin-specific logic is divided into the *Plugin* and *UI* packages. These packages contain logic to start up the plugin and act as a controller, and run the user interface through button placement and creation of sub-windows, respectively.

The second and most important element is the domain. The domain consists of two sets of layers, indicated by solid and dotted lines. The layers drawn with a solid border are layers that correspond to (parts of) the model as described in Chapter 5. These layers are very strict in that they can only use elements from themselves or lower layers, but never higher layers. Conversely, the layers drawn with a dotted border are supporting layers that can be called from any point.

[1] Eclipse version 2023-06 was used

[2] ReFD can be accessed through the following GitHub repository: https://github.com/NHLStenden-ISAL/ReFD

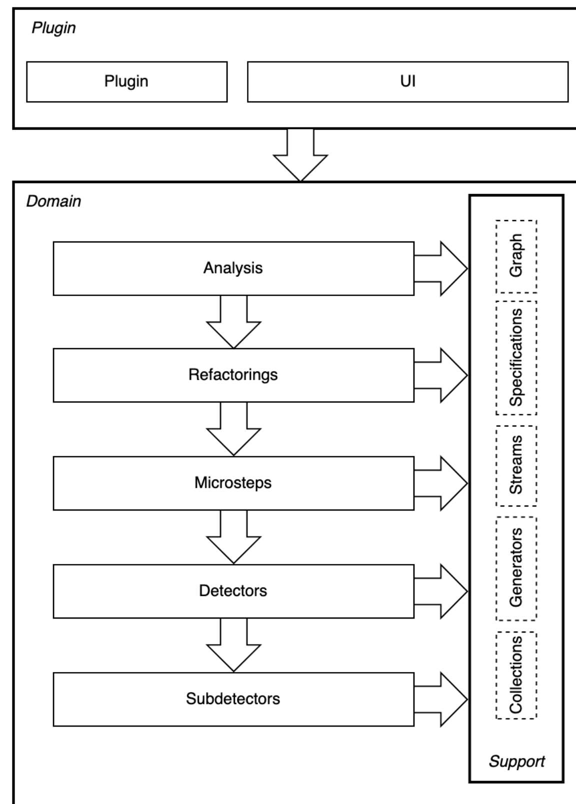


Figure 6.1: Architecture of the Eclipse plugin

6.2. PROGRAM GRAPH

To diagnose an intended refactoring, we need some way of gaining insight into the codebase the refactoring is to be applied to. We do this by converting the codebase into a graph that looks like an Abstract Syntax Tree (AST) but is augmented with a data-flow analysis. This means its nodes do not only have an edge connecting them to their parent, but can have multiple outward and inward edges. To convert the codebase into such a graph, we use an existing tool called Atlas[3]. The adapter design pattern [5] is applied to the graph Atlas uses, which wraps its elements used by our developed plugin, thereby creating our own version that we will call *program graph*. While the public interface of these wraps looks a lot like that of the objects they wrap, they contain only the minimum of functionality to make the refactoring diagnosis possible. We chose this approach because it has two advantages: the wrapping classes around the Atlas elements form a minimal model of the graph facilities necessary, and it becomes easier to reimplement the program graph if need be.

Each node in the program graph is of the type `ProgramLocation`, and represents some location in the codebase. Because these *program locations* are part of the program graph, they are linked by edges of that graph. Each edge in the program graph is of the type `Relation`. *Relations* and program locations have a specific type, which is bestowed upon them in the form of a tag. This manner of weak typing is used because the graph model is rather abstract, and its only two strong types are `ProgramLocation` and `Relation`. Atlas handles tagging by using strings, but our software uses specific enumerations to guarantee

[3] https://www.ensoftcorp.com/atlas/

appropriate tagging. To work with strongly typed program locations, collections must be used. These will be explained in Section 6.7.

Finally, the program graph contains a facility to query the graph: the `GraphQuery`. This too is an adapter that wraps Atlas's query facility called `Q`. From one or more starting nodes, `GraphQuery` can walk the program graph.

Example 6.2.1 Suppose we have two nodes m and m' representing methods, and a relation of the *override* type defined from m to m'. This means method m overrides method m'. If a `GraphQuery` has m as a starting node, it can query the relations present and walk the override relation from m to m'. `GraphQuery` can be thought of to contain a set of program locations as results of a query. The `GraphQuery` started with the set $\{m\}$. The subsequent query then altered the set of program locations. This meant that after walking the override relation, the `GraphQuery` contained the set $\{m'\}$. □

Sometimes, however, we need to work with a program location that might not exist (yet). As an example, this can happen when adding a new method. To be able to convey its location before the method is added to the codebase or program graph, some medium other than `ProgramLocation` must be used because that type of program location always exists in the graph.

6.3. REPRESENTING POSSIBLY NONEXISTENT CODE LOCATIONS

To model program locations that possibly do not exist, we use program location *specifications* (called *location specification* from here on). A location specification is an object that contains a description of all defining features of some syntax construct.

Example 6.3.1 Consider the code displayed in Figure 4.2. If we want to describe the method `toString()`, we use the code in Figure 6.2.

```
1  ClassSpecification employee_spec = //creation logic here...
2  MethodSpecification toString_spec =
3     new MethodSpecification(
4        "toString", //method name
5        new ArrayList<ParameterSpecification>(), //parameters
6        AccessModifier.PUBLIC, //visibility
7        false, //static
8        false, //abstract
9        "void", //return type
10       employee_spec //parent class
11 );
```

Figure 6.2: MethodSpecification usage example for `Employee.toString()`

□

The specification shown in Example 6.3.1 specifies a method that actually exists in the codebase. This means we can find a corresponding program location in the program graph for it. We can however easily change this specification to mean a method that does not exist. For example, if we change the string on line 4 of Figure 6.2 from `"toString"` to `"toString2"`, the specification now specifies a nonexistent method.

Now that we know how to represent existing program locations, as well as nonexistent ones, we can start to look at how a refactoring is implemented.

6.4. REFACTORING

A refactoring is modeled by the abstract class `Refactoring`, which provides facilities every refactoring needs. The most important of these facilities is the ability to add microsteps to it.

Example 6.4.1 In this example, we will review the implementation of the Pull Up Method refactoring. Pulling up a method implies that a method should be moved from one class to one of its superclasses, as shown in Figure 6.3.

```
1  public class PullUpMethod extends Refactoring {
2
3     private final MethodSpecification target;
4     private final boolean toDirectSuperclass;
5
6     public PullUpMethod(MethodSpecification target, ClassSpecification destination) {
7        //...
8
9        MethodSpecification newLocation = target.copy();
10       newLocation.setEnclosingClass(destination);
11
12       microstep(new MoveMethod(target, newLocation));
13    }
14
15    //...
16
17 }
```

Figure 6.3: PullUpMethod.java

□

The class `PullUpMethod` shown in Example 6.4.1 represents a Pull Up Method refactoring. Refactorings contain microsteps, and this particular refactoring contains the `MoveMethod` microstep. We note that the code for the refactoring is rather declarative. It only specifies the constituent elements of the refactoring and no analysis logic needs to be specified here. For `PullUpMethod`, this is done by a `MoveMethod` microstep, which is part of the refactoring analysis decomposition.

6.5. MICROSTEP

To decompose refactorings, microsteps are used. A microstep is modeled by the abstract class `Microstep`, which provides facilities every microstep needs. Because, as described in Chapter 5, a microstep carries one or more potential risks, `Microstep` provides the ability to add detectors to it that determine if a potential risk is present in the codebase.

Additionally, a microstep can also be composed of other microsteps. This is modeled by the abstract class `CompositeMicrostep`. The `MoveMethod` microstep we saw in Example 6.4.1 is such a microstep, and consists of microsteps `AddMethod` and `RemoveMethod`, which are part of the greater library of microsteps.

Example 6.5.1 shows the `AddMethod` microstep carrying multiple potential risks, which are modeled by detectors. They can be added by the provided `potentialRisk(...)` method, which, similarly to the refactoring, makes microsteps declarative.

Example 6.5.1 To illustrate how a microstep works, we will focus on the `AddMethod` microstep displayed in Figure 6.4.

```
1  public class AddMethod extends Microstep {
2
3     private final MethodSpecification methodToAdd;
4
5     public AddMethod(MethodSpecification methodToAdd) {
6        this.methodToAdd = methodToAdd;
7
8        potentialRisk(new DoubleDefinition.Method(methodToAdd));
9        potentialRisk(new BrokenSubTyping.Method(methodToAdd));
10       potentialRisk(new CorrespondingSubclassSpecification.Method(methodToAdd));
11       potentialRisk(new OverloadParameterConversion.Method(methodToAdd));
12    }
13
14    //...
15
16 }
```

Figure 6.4: AddMethod.java

□

6.6. DETECTOR

A detector determines if a potential risk is present in the codebase. A detector is modeled by the abstract class `Detector`. This class is generic and accepts a type parameter that signifies the type of program location set the detector outputs. The `Detector` class is not fully implemented, and method `actualRisks()` needs to be implemented its concrete subclasses.

Example 6.6.1 To explain how a detector works, we show how `DoubleDefinition`, the first of the three detectors seen in Example 6.5.1, is implemented in Figure 6.5.

```
1  public final class DoubleDefinition {
2
3     public static class Method extends Detector<MethodSet> {
4
5        private final MethodSpecification subject;
6
7        //...
8
9        @Override
10       public MethodSet actualRisks() {
11          return new ProgramComponentsGenerator()
12                .stream()
13                .classes()
14                .classesByName(subject.getEnclosingClass().getClassName())
15                .methods()
16                .methodsWithSignature(subject.getMethodName(),
17                                      subject.getParameterTypes()
18                ).collect();
19       }
20
21    }
22
23 }
```

Figure 6.5: DoubleDefinition.java

□

Detectors have specific versions depending on the syntax construct they are applied to. These versions are modeled as inner classes that are grouped in an enclosing class representing the type of detector. This means that the detector shown in Example 6.6.1 is a `DoubleDefinition` detector, specifically for a method. It receives a `MethodSpecification` which is the method to be added to a certain context. This detector checks if, were that to happen, a double definition of the method would take place.

The example detector implements the method `actualRisks()`, which determines the actual risks for this detector based on the given `MethodSpecifiation` and its program context. This method is implemented using several subdetectors. These subdetectors are chained together in the form of a stream, similar to Java streams. As a result, `actualRisks()` returns a set of methods called `MethodSet`.

6.7. SETS, STREAMS & GENERATORS

To be able to ask questions about the codebase a refactoring is to be applied to, we need a system to facilitate this. In Section 6.2, `GraphQuery` might seem to solve this issue. However, in this context, a serious limitation of the graph model is that it is not rich enough to deal with questions specific to a syntax construct. If we were to try and implement those, it would result in `GraphQuery` containing all manner of queries meant for a vast range of types of program locations. This would lead to the possibility of a query that first queries the superclasses of a class, after which it queries their return type. This last query is clearly meant for a method, and we need a model that is rich enough to be able to determine which queries are valid, and which are not.

Our resulting model consists of strongly typed sets of program locations, corresponding streams, and generators. Each of these elements will be explained below briefly.

SETS

To be able to determine which queries can be executed on a program location, we need to know its type. We facilitate this by enclosing the program location in a set that signifies this type. This has the advantage that queries are not only able to be executed on just one element, but we can query the entire set of program locations. Examples of these sets are `MethodSet`, `ClassSet` and `FieldSet`.

GENERATORS

Generators are a facility that can generate a set of program locations. An example of a generator can be seen on line 11 of Figure 6.5. This `ProgramComponentsGenerator` generates a set of all program components. These are the highest components of the program, which in this case are its compilation units. Simply put, generators provide a way to quickly access the program locations representing the codebase.

STREAMS

Now that we have typed sets of program locations, we can start to apply queries to these sets. From each type of set or generator, a stream of the same type can be created. For example, from a `MethodSet` a `MethodStream` can be created. When creating a stream from a set of program locations, this set becomes the new stream's source. These streams contain methods that apply queries on the stream, not on the source of the stream per se. Streams

have an internal list of queries to be applied to them, and when a method to apply a query is called on the stream, it copies itself and adds the query to the internal list of that copy. In this way, streams are immutable. They form a sequence of queries, which are only applied once a terminal method is called on the stream. For example, asking the stream to collect all results, as is done on line 18 of Figure 6.5, triggers the stream to apply its queries. When this happens, the first query is applied to the stream's source, but the subsequent queries are applied to the output of a previous query.

Example 6.6.1 shows us how building a query works. On lines 11 and 12 of Figure 6.5, we see a stream being created from a generator. We note that at this point, the stream is a `ProgramComponentsStream`. This stream supports the query to get all classes contained in the stream. In turn, we can call the method `.classes()` on the stream. Note that this does not execute a query. Rather, it creates a new stream and adds the query to the internal list of the stream. This new stream is of type `ClassStream`, and only query methods appropriate for classes can be called on it. In this way, a situation in which a method gets queried for its superclass cannot occur.

The queries added to the stream are part of a library of what are called subdetectors.

6.8. SUBDETECTOR

A subdetector is a part of a query used by a detector to determine if a potential risk is present in the codebase. This means that a subdetector is only part of that process, so its responsibilities are not as great. A subdetector receives a set of program locations, applies one or more queries, and returns its results as a new set of program locations.

A subdetector is modeled by the abstract class `Subdetector`. `Subdetector` is not fully implemented. Its method `applyOn(Set<ProgramLocation>)` needs to be implemented by every concrete subclass.

Example 6.8.1 We start to show how a subdetector works by using `Overrides` as an example. This subdetector can be seen in Figure 6.6.

```
1  public final class MethodSubdetectors {
2     //...
3
4     public static class Overrides extends Subdetector {
5
6        @Override
7        public Set<ProgramLocation> applyOn(Set<ProgramLocation> locations) {
8           GraphQuery gq = Graph.query(locations);
9
10          return gq.descendantsOn(
11                gq.universe()
12                  .relations(Tags.Relation.OVERRIDES)
13          ).locations(Tags.ProgramLocation.METHOD).locations();
14       }
15
16    }
17
18    //...
19
20 }
```

Figure 6.6: MethodSubdetectors.java

□

Like detectors, streams are also grouped by type. In Example 6.8.1, we find one of these groupings to be `MethodSubdetectors`. This means that all subdetectors contained in this class expect methods to be supplied to them

To explain the inner workings of a subdetector, we will look at the more elementary `Overrides` subdetector from Example 6.8.1. The `Overrides` subdetector returns the methods that are overridden by the methods supplied to the `applyOn(...)` method. It does this by executing a query on the program graph, shown as `Graph.query(...)`. The resulting `GraphQuery` object can walk the program graph and in this case, traverse relations in the graph tagged with the `Tags.Relation.OVERRIDES` tag. To be certain of the result, it then filters the found locations for methods and lastly returns the result.

6.9. ANALYSIS

The previous sections explained that parts of the domain are developed in a declarative manner. A refactoring forms a tree with the refactoring at its root, microsteps as internal nodes, and detectors as its leaves. This tree does not contain logic to manage the danger analysis process. Instead, this is implemented in a visitor that visits the refactoring's tree structure. This visitor called `DangerAnalyser` walks the tree and aggregates all detector results. However, these results might contain false positives. To remove these, it uses the refactoring's verdict function.

6.10. VERDICT

To filter false positive results from detectors, a verdict specific to each refactoring is constructed. A base class for such a verdict exists, and is called `VerdictFunction`. `VerdictFunction` is an abstract class that uses double dispatch in the same way a visitor might. For every detector, `VerdictFunction` contains a default visit method that lets every result through. If a detector result should be filtered, its corresponding visit method should be reimplemented. When letting results through, they are labeled with the detector that found them.

In Example 6.10.1, we see that this verdict function has a specific implementation for the detector `RemovedConcreteOverride.Method`. This means that when `DangerAnalyser` receives results from that detector and passes them to this method, those results are filtered based on its implementation. In this case, we see that it checks if the refactoring pulls the method up to the direct superclass or not. In case we pull up to the direct superclass, we remove the detector's results on line 15 by `none(detector)`. In all other cases, we let all results through on line 18 by `all(detector)`.

Example 6.10.1 To explain how the verdict function works, we look at the verdict function for the `PullUpMethod` refactoring. The parameters on lines 6 and 8 were removed for brevity.

```
1  public class PullUpMethod extends Refactoring {
2
3     //...
4
5     @Override
6     public VerdictFunction verdictFunction(...) {
7
8        return new VerdictFunction(...) {
9           //...
10
11          @Override
12          public void visit(RemovedConcreteOverride.Method detector)
13          {
14             if (toDirectSuperclass) {
15                none(detector);
16             }
17             else {
18                all(detector);
19             }
20          }
21
22          //...
23
24       };
25
26    }
27
28 }
```

Figure 6.7: VerdictFunction for PullUpMethod.java

□

6.11. CHANGING THE GRAPH

In some cases, a refactoring is more complex because it adds elements to the code base that previously did not exist. The *Combine Methods into Class* refactoring is such a case because before it moves methods to a class, it first creates that class anew. At first sight, this is rather problematic when working with the program graph, because it only contains code that actually exists at this moment. Therefore, when a method is moved to a new class that does not exist, all queries on the graph will not work correctly.

Example 6.11.1 Suppose we move a method `toString()` to a class to be newly created, and next we move another method `toString()` to that same class. In this case, the system will not detect a double definition of `toString()` at that location because the class does not exist in the graph. Interestingly, even if the class did exist, the system would still not detect a danger. This is because for this to happen, the first `toString()` needs to be already present in the class for the double definition to trigger. However, this is not the case. □

This can be remedied by having the microsteps of a refactoring augment the program graph. As an example, this would mean that the microstep `AddClass` adds a class node to the program graph and adds the correct relations to it. This functionality has been implemented in the microsteps.

6.12. LIST OF IMPLEMENTED MICROSTEPS AND DETECTORS

To be able to support both the Pull Up Method and Combine Methods into Class refactorings, several microsteps have to be implemented. Pull Up Method requires us to move a target method from its enclosing class to its destination class. This results in a *MoveMethod* microstep. A MoveMethod is a combination of two microsteps: *AddMethod* to add the method to the destination class, and *RemoveMethod* to remove the method from its original enclosing class.

We note here that the choice to combine AddMethod and RemoveMethod into the microstep MoveMethod is not a trivial decision, and is based on the fact that microsteps have specific potential risks associated with them, as explained in Chapter 5. MoveMethod has a specific potential risk associated with it that is not associated with its constituent microsteps. This will be explained in the following sections.

The Combine Methods into Class refactoring also moves methods from one class to another, which means we can reuse our previously found MoveMethod microstep. The methods are moved to a newly created class, which means we need a microstep to create a new class: *AddClass*. This analysis leads us to the following list of microsteps: AddMethod (AM), RemoveMethod (RM), MoveMethod (MM), and AddClass (AC).

The microsteps found correspond to the work of Verduin [21], in which he also provides a preliminary description of hazards associated with those microsteps. These hazards can be thought of as potential risks because they have the same form. Working from these potential risks as a base, we added multiple additional potential risks. Below, we will list for each microstep the associated potential risks and their corresponding detector. Each combination of potential risk and detector is labeled with two letters corresponding to the list of microsteps above, and an index number.

ADDMETHOD

Table 6.1: Potential risks for AddMethod microstep and associated detectors

Label	*Detector & Potential Risk*
AM-1	DoubleDefinition - Method to add is already defined in context of target location
AM-2	BrokenSubTyping - Method to add is defined in superclass of context as well, thus overriding that method in context
AM-3	CorrespondingSubclassSpecification - When adding a method, and a method with the same signature exists in one or more subclasses, its specification might not correspond to these existing methods
AM-4	OverloadParameterConversion - Method to add overloads existing method but parameter types precede in automatic type conversion rules, thus leading to previous calls to the existing method now calling the new method unintentionally

REMOVEMETHOD

Table 6.2: Potential risks for RemoveMethod microstep and associated detectors

Label	*Detector & Potential Risk*
RM-1	MissingDefinition - Method still called after removal
RM-2	RemovedConcreteOverride - Override method is removed, thus defaulting to super implementation
RM-3	LostSpecification - When method is removed, methods overriding it will not have a specification anymore to adhere to
RM-4	MissingSuperImplementation - Method to be removed is concrete and not all of the containing class's subclasses have an override implementation, thus leading to possible missing implementations
RM-5	MissingAbstractImplementation - Method to be removed implements abstract method in concrete class, while its implementation is mandatory

MOVEMETHOD

The MoveMethod microstep includes all potential risks of AddMethod and RemoveMethod, which are not listed here.

Table 6.3: Potential risks for MoveMethod microstep and associated detectors

Label	*Detector & Potential Risk*
MM-1	BrokenLocalReferences - Method body contains references to elements from the local context (current class and superclasses), such as fields and methods

ADDCLASS

Table 6.4: Potential risks for AddClass microstep and associated detectors

Label	*Detector & Potential Risk*
AC-1	DoubleDefinition - Class to add is already defined in context of target location

6.13. PLUGIN DEMONSTRATION

In this section, we present the prototype plugin which shows the execution of the diagnosis process for the Pull Up Method refactoring. Figure 6.8 shows the normal setup of the tool. The refactoring case study example is loaded as an example case. We will show how the Pull Up Method refactoring works in the current version of the tool. To do this, we must first select a method to pull up. In Figure 6.8, this has already been done, and the method `salaryBonus(int)` is selected.

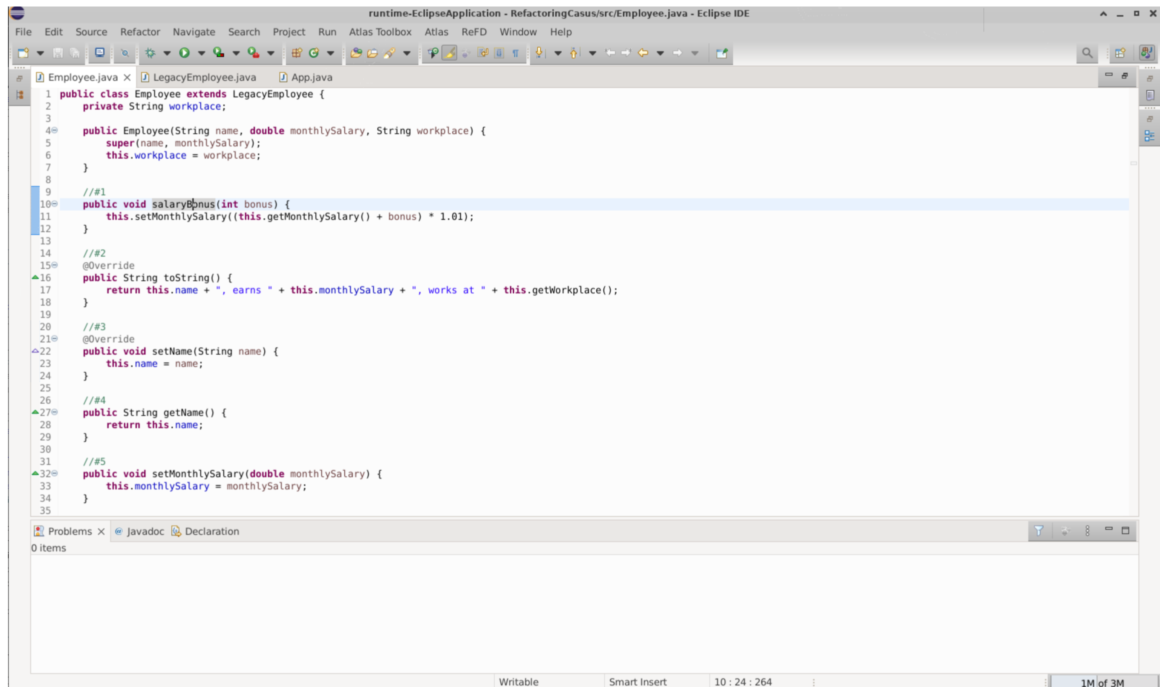


Figure 6.8: The tool with the refactoring case study example loaded

To initiate the refactoring diagnosis process, the user can select the correct refactoring from the menu as shown in Figure 6.9.

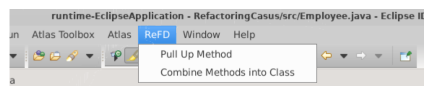


Figure 6.9: Refactoring selection menu

After selecting the Pull Up Method refactoring entry from the menu, the tool will check if a method is currently selected in the editor. If this is not the case, an error message will be displayed. In this case, because we did select a method, the tool now presents us with a selection screen, as shown in Figure 6.10. This screen contains all superclasses of the method's enclosing class, in this case superclasses of `Employee`. From this list, a superclass must be selected to start the diagnosis process.

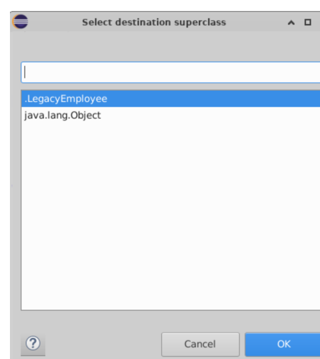


Figure 6.10: Selecting destination class

We select superclass `LegacyEmployee`. Next, the tool will start the analysis process in a different thread to prevent the user interface from lagging during the process. Once the results are in, they are displayed in Eclipse's *Problems* window, as shown in Figure 6.11.

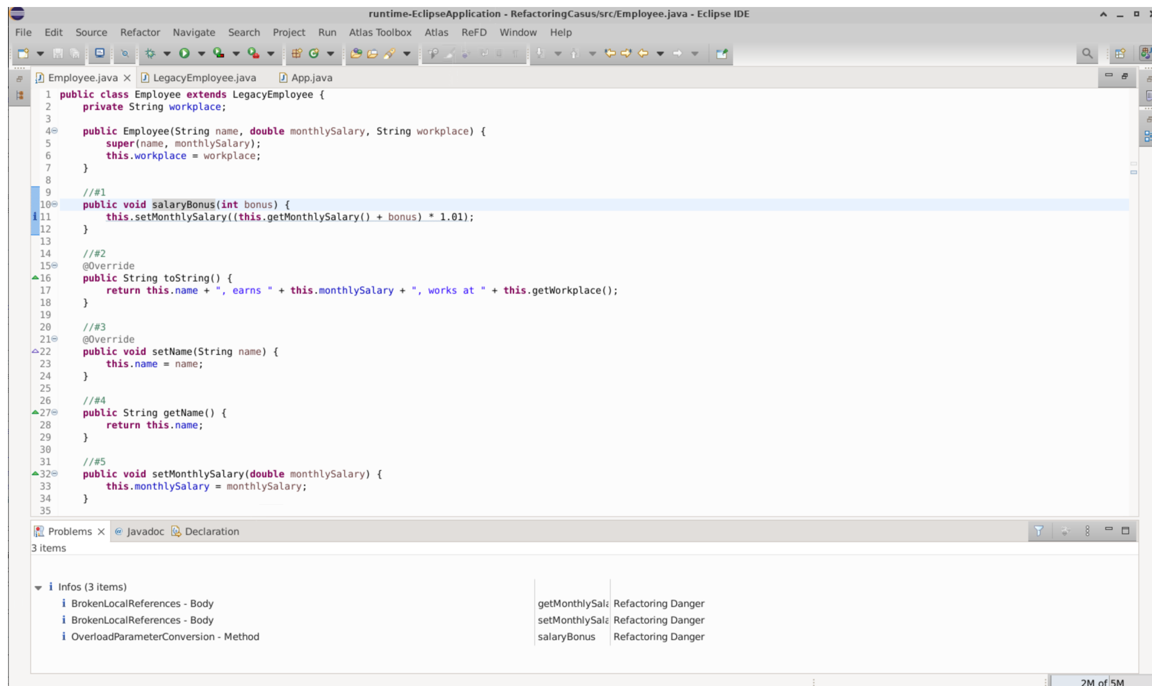


Figure 6.11: Results from refactoring showing in Problems window

7
VALIDATION

In this chapter, we provide a small validation of the developed tool by way of six example results of the supported refactorings. The case study example presented in Chapter 4 is used as the subject of the refactorings. The Pull Up Method refactoring is validated with four examples that are representative of the problems that can be encountered. The Combine Methods into Class refactoring is validated with two examples, for which two sets of methods were selected to move to a new class. We do note that this is not an extensive validation, but only an indication that the tool and, more importantly, its model and design work. Extensive validation is more difficult and is part of the future work described in Chapter 10.

7.1. PULL UP METHOD

We validate the tool's implementation of Pull Up Method on four methods found in class `Employee`: `salaryBonus(int)`, `toString()`, `setName(String)`, and `getWorkplace()`.

METHOD EMPLOYEE.SALARYBONUS(INT)

Table 7.1 details the dangers determined for the Pull Up Method refactoring applied to `Employee.salaryBonus(int)`. These dangers were determined by manually checking if potential risks associated with the refactoring's constituent microsteps are present in the code context and should not be filtered by the verdict function. The developed plugin was tested on this case and yielded the same results.

Table 7.1: Dangers determined for Pull Up Method applied to `Employee.salaryBonus(int)`

Detector	*Reason*
BrokenLocalReferences	`salaryBonus(int)` contains local references when calculating the salary bonus and setting the new salary value
OverloadParameterConversion	When adding `salaryBonus(int)` to `LegacyEmployee`, it overloads `salaryBonus(double)`, and the integer parameter variant is narrowing in relation to the double parameter variant

METHOD EMPLOYEE.TOSTRING()

Table 7.2 details the dangers determined for the Pull Up Method refactoring applied to `Employee.toString()`. These dangers were determined by manually checking if potential risks associated with the refactoring's constituent microsteps are present in the code context and should not be filtered by the verdict function. The developed plugin was tested on this case and yielded the same results.

Table 7.2: Dangers determined for Pull Up Method applied to `Employee.toString()`

Detector	*Reason*
DoubleDefinition	A method with the signature `toString()` is already present in `LegacyEmployee`
BrokenSubTyping	`toString()` is already defined in superclass of `Employee`
BrokenLocalReferences	`toString()` in `Employee` uses local references while producing the string value

METHOD EMPLOYEE.SETNAME(STRING)

Table 7.3 details the dangers determined for the Pull Up Method refactoring applied to `Employee.setName(String)`. These dangers were determined by manually checking if potential risks associated with the refactoring's constituent microsteps are present in the code context and should not be filtered by the verdict function. The developed plugin was tested on this case and yielded the same results.

Table 7.3: Dangers determined for Pull Up Method applied to `Employee.setName(String)`

Detector	*Reason*
DoubleDefinition	A method with the signature `setName(String)` is already present in `LegacyEmployee`

METHOD EMPLOYEE.GETWORKPLACE()

Table 7.4 details the dangers determined for the Pull Up Method refactoring applied to `Employee.getWorkplace()`. These dangers were determined by manually checking if potential risks associated with the refactoring's constituent microsteps are present in the code context and should not be filtered by the verdict function. The developed plugin was tested on this case and yielded the same results.

Table 7.4: Dangers determined for Pull Up Method applied to `Employee.getWorkplace()`

Detector	*Reason*
BrokenLocalReferences	`getWorkplace()` in `Employee` uses local references while producing a string value

7.2. COMBINE METHODS INTO CLASS

We validate the tool's implementation of Combine Methods into Class on two sets of methods found in classes `Employee` and `LegacyEmployee`: `Employee.toString()` and `LegacyEmployee.toString()`, and `Employee.getWorkplace()` and `Employee.setWorkplace(String)`.

METHODS EMPLOYEE.TOSTRING() AND LEGACYEMPLOYEE.TOSTRING()

Table 7.5 details the dangers determined for the Combine Methods into Class refactoring applied to `Employee.toString()` and `LegacyEmployee.toString()`. These dangers were determined by manually checking if potential risks associated with the refactoring's constituent microsteps are present in the code context and should not be filtered by the verdict function. The developed plugin was tested on this case and yielded the same results.

Table 7.5: Dangers determined for Combine Methods into Class applied to `Employee.toString()` and `LegacyEmployee.toString()`

Detector	*Reason*
DoubleDefinition	Because both methods have the same signature, they conflict in the new class
MissingSuperImplementation	When both methods are moved, class `Employee` loses its own implementation of `toString()`, but also the implementation in superclass `LegacyEmployee`
BrokenLocalReferences	Both methods use multiple local references
RemovedConcreteOverride	When the methods are moved, they trigger this detector because they are both concrete and overrides of another method

METHODS EMPLOYEE.GETWORKPLACE() AND EMPLOYEE.SETWORKPLACE(STRING)

Table 7.6 details the dangers determined for the Combine Methods into Class refactoring applied to `Employee.getWorkplace()` and `Employee.setWorkplace(String)`. These dangers were determined by manually checking if potential risks associated with the refactoring's constituent microsteps are present in the code context and should not be filtered by the verdict function. The developed plugin was tested on this case and yielded the same results.

Table 7.6: Dangers determined for Combine Methods into Class applied to `Employee.getWorkplace()` and `Employee.setWorkplace(String)`

Detector	*Reason*
BrokenLocalReferences	Local reference to `workplace` is present
MissingDefinition	a call to `getWorkplace()` is present in the class context

8
DISCUSSION

In this chapter, we discuss the research process in two ways. First, we explain some points of interest that presented themselves during research but did not appear in the final prototype. Second, we briefly discuss the limitations of this study.

8.1. GENERAL TOPICS OF INTEREST

One of the major contributions of this study is the supporting layers in the architecture discussed in Section 6.1. However, it was not clear how program locations could best be represented. From the start, it was clear that program locations that exist are also nodes within the program graph. However some program locations did not always exist, such as the location of a method that still has to be created. Multiple ways of dealing with this problem were reviewed. One such way was to use XPath[1] as a query and description language for program locations. An XPath-like implementation to describe program locations was created to investigate if this would work. At the start, it did, but mainly to select pieces of code to apply a refactoring on. Such XPath queries look like textual query descriptions, and as research progressed, the description element became more important. This led to disregarding the XPath support, but the notion of a location description was kept in order to create the Specification layer now present in the architecture.

Another important contribution is the system of typed sets of program locations and their streams. This system was introduced as a medium to facilitate the chaining of subdetectors and create a uniform way to think about this chaining. Earlier during development, and in the work of Wernsen [22], the graph query facilities and subdetectors were mixed in together. However, creating a clear distinction between these two was important to understand better what functionality a subdetector should provide to a detector, and what functionality graph queries should provide to subdetectors. Paramount here is that a subdetector abstracts away the notion of moving through a graph. Instead, while working with subdetectors, instead of handling program locations, we handle typed sets of program locations. This means that we know the type of program location we are dealing with, such as a set of methods, which we then can use a subdetector on, such as one that results in a set of classes the methods are enclosed in. This greatly enhances both the clarity of the detector implementations, as well as their reliability.

[1] https://www.w3.org/TR/1999/REC-xpath-19991116/

The tool resulting from this research has several advantages over previously developed prototypes. Earlier prototypes developed during the research described in Section 2.1 did not implement the model (fully). In addition, the current version of the tool also adheres to the open-closed principle, in the sense that it heavily uses abstract base classes to provide a framework that can easily be extended.

We note that it was difficult to accurately determine the potential risks for each of the microsteps present in the plugin implementation. During our research, it seemed like the potential risks could only be determined by reviewing all the possible situations a microstep could be used in and how it affects behavior preservation. It is hard to say if this is possible to do accurately, which is even more clear when looking at how to validate if truly every potential risk was found.

Concerning the verdict function, our implementation allows for a verdict function per refactoring. The verdict function is implemented per detector. This means that within a single refactoring, for two different microsteps using the same detector, no alternative verdict can be specified for that detector. Put in other words, per refactoring, there is only one type of verdict available for each detector. When detectors are reused, problems can arise.

During our study, the manner of gathering all relevant information from the refactoring diagnosis process was also investigated. During this process, a Root Cause Analysis (RCA) was briefly considered as a data structure for these results. However, this was dropped because an RCA is a tree structure, which it turned out was also the case for the combination of a refactoring and its microsteps, and detectors.

8.2. LIMITATIONS OF THE STUDY

A clear limitation of this study is the size and complexity of the case study that was used to review the dangers of the Pull Up Method refactoring - more specifically, the potential risks of its microsteps and its verdict function. As noted in Section 8.1, currently, potential risks are found by reviewing situations microsteps can be used in. Because the reviewed case study was small and relatively simple, this might lead to the list of found potential risks, and in turn the verdict functions, to be incomplete.

In relation to the previously mentioned limitation, only two refactorings were reviewed and this might also negatively impact the study results. A greater number of reviewed refactorings increases the number of reviewed situations, and a greater number of reviewed situations increases the potential risks found.

The way the implementation was tested could be improved. As described in Chapter 7, this was done by validating the tool's results against the case study example from Chapter 4. We suggest two improvements. First, if the form of potential risks could be better formalized, the testing regime for the tool could perhaps be as well. Second, such formalized potential risks could be used to create a test suite which the tool can be validated against. In this way, changes in the system can be shown not to negatively impact its ability to produce results.

Finally, the case study from Chapter 4 was only reviewed by six participants. While a mix of students and experts was found, the results were, in general, quite similar. This might mean that we found a general way of handling such refactorings, but it might also mean that the participants simply had a similar view by chance. Increasing the number of participants would probably yield different views on the refactorings.

9 Conclusion

In this thesis, we presented a new model to detect refactoring dangers and its first implementation in the form of an Eclipse plugin called ReFD. The model detects dangers by splitting a refactoring up into microsteps that have associated potential risks. A detector examines the code context to determine if the potential risk is present in the code, thus becoming an actual risk. The resulting actual risks are evaluated by a verdict mechanism to remove false positives. The final result comprises labeled program locations for which warnings can be given.

The Eclipse plugin resulting from this research has several advantages over previously developed prototypes. Earlier prototypes developed during the research described in Section 2.1 did not implement the model (fully). In addition, the current version of the tool also adheres to the open-closed principle, in the sense that it heavily uses abstract base classes to provide a framework that can easily be extended.

The Eclipse plugin has been shown to be able to detect dangers in a refactoring by closely implementing the aforementioned model. It supports two refactorings, *Pull Up Method* and *Combine Methods into Class*, that are based on Fowler's descriptions [4] and the results of the case study presented in Chapter 4. In this case study, we reviewed the problems students and experts encounter while performing multiple Pull Up Method refactorings.

A refactoring contains one or more microsteps that produce actual risks, and a verdict function that filters results from the microsteps. The implemented microsteps *AddMethod*, *RemoveMethod*, *MoveMethod* and *AddClass* and their potential risks are based on the work of Verduin [21] and the results of the case study. The detectors were developed based on the previously found potential risks and were implemented using subdetectors through *streams*.

Streams were developed to chain subdetectors and ensure they receive the correct type of program locations. Multiple program locations can be combined as typed sets of program locations, such as a `MethodSet` or `ClassSet`. From a typed program location set, a typed stream can be created, such as a `MethodStream` or `ClassStream` respectively. These streams can process the program locations inside by chaining subdetectors as operations on the stream, much like how Java streams can form a chain of operations. Each type of stream can only use subdetectors that are suitable for the type of program locations inside them.

Subdetectors receive a set of program locations to process. They query the code context based on the program locations they receive and produce their result as a set of program

locations as well. This is also what allows the subdetectors to be chained together because the output of one subdetector can be the input of another.

A subdetector can query the code context through a graph representation of the code the refactoring is being applied to. In our plugin implementation, the graph is created by an external tool called Atlas. The program locations that subdetectors receive are simply nodes in this graph representation. This means that a program location always exists within the graph representation.

Sometimes, a location in a program does not exist yet, for instance before adding a class, that class is not yet a node in the program graph. When dealing with such locations, we use *location specifications*. A location specification is simply an object containing a description of the identifying elements of a program location. As an example, this means that a `ClassSpecification` contains the class's name and the package it belongs to. A location specification can be used to search for this location in the program graph or to create it if it does not exist.

Finally, the implementation was validated against the refactoring example case from Chapter 4. Both refactorings were tested on multiple inputs and yielded the expected results.

10

Future Work

This research has shown a lot of potential for future research projects. In particular, it resulted in ReFD, a platform for future development of refactoring diagnosis theory and practice. However, a number of things are still missing.

The current refactoring diagnosis model allows for refactoring danger detection to be built in a modular way. We want to further the development of the tool and build a more extensive library of refactorings. This will give us more insight into the way the refactorings can be split up and will help us list the basic components of refactoring advice. In addition, it might also improve the architecture of our model.

However, when building a more extensive library of refactorings, the problem of determining if all potential risks were found during analysis, as described in Section 8.1, will rear its head again. We must gain more insight into a systematic process to find potential risks for microsteps and a way to determine if our findings are complete.

A more general problem that needs additional research is how a systematic way can be found to transform a refactoring from Fowler's work [4] into a list of microsteps, as this is currently left up to the insightfulness of the researchers.

The system currently only detects dangers, but cannot generate refactoring advice yet. This advice system will probably use some kind of tree structure. It might be possible to determine the nodes in the tree based on the dangers that are found and find leaves of that tree when a subsequent refactoring no longer produces dangers. We want to explore the way the current system can be expanded by the addition of refactoring advice generation.

The way the program graph is currently changed is not a very structured process. It could be worthwhile to formalize this aspect of microsteps better so, just like the detection mechanism, a more formal specification of graph changes can be given.

As a final future work, we want to make it easy to compose new refactorings to provide advice on. As such, we propose a tool that allows the user to visually build a refactoring from basic building blocks which will then be able to generate advice just like the refactorings already present.

Bibliography

[1] de Beer, P. *Code Context Based Generation of Refactoring Guidance*. Open Universiteit, 2019. Master's Thesis. 2, 7

[2] Evans, E. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004. 12

[3] Fowler, M. *Refactoring*. Addison-Wesley Professional, 2018. 8

[4] Fowler, M., Fraser, S., Beck, K., Caputo, B., Mackinnon, T., Newkirk, J., and Poole, C. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. 1, 2, 3, 12, 13, 21, 49, 51

[5] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, Reading, MA, USA, 1995. 30

[6] Haendler, T., Neumann, G., and Smirnov, F. An interactive tutoring system for training software refactoring. *Instructor 1* (2019), 4. 9

[7] Hilberink, H. *Contextual Refactoring - Towards Risk-Driven Fowler based Refactoring Guidance*. Open Universiteit, 2021. Master's Thesis. 7

[8] Kim, M., Zimmermann, T., and Nagappan, N. An empirical study of refactoring challenges and benefits at microsoft. *IEEE Transactions on Software Engineering 40*, 7 (2014), 633–649. 8

[9] Kniesel, G., and Koch, H. Static composition of refactorings. *Science of Computer Programming 52*, 1-3 (2004), 9 – 51. Special Issue on Program Transformation. 8

[10] Mens, T., and Tourwé, T. A survey of software refactoring. *IEEE Trans. Softw. Eng. 30*, 2 (Feb. 2004), 126–139. 1, 2, 8

[11] Mongiovi, M., Gheyi, R., Soares, G., Ribeiro, M., Borba, P., and Teixeira, L. Detecting overly strong preconditions in refactoring engines. *IEEE Transactions on Software Engineering 44*, 5 (August 2018), 429–452. 2

[12] Murphy-Hill, E., Parnin, C., and Black, A. P. How we refactor, and how we know it. *IEEE Transactions on Software Engineering 38*, 1 (2012), 5–18. 8

[13] Oates, B. J. *Researching Information System and Computing*. Sage Publications Ltd, London, England, 2006. 12

[14] Opdyke, W. F. *Refactoring object-oriented frameworks*. University of Illinois at Urbana-Champaign, 1992. 1, 2, 8

[15] PASSIER, H., BIJLSMA, L., AND BOCKISCH, C. Maintaining unit tests during refactoring. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (New York, NY, USA, 2016), PPPJ '16, ACM, pp. 18:1–18:6. 2

[16] PROIETTI, M., AND PETTOROSSI, A. Semantics preserving transformation rules for prolog. *ACM SIGPLAN Notices 26*, 9 (1991), 274–284. 1

[17] SOARES, G. Making program refactoring safer. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2* (2010), ACM, pp. 521–522. 2, 8

[18] TOKUDA, L., AND BATORY, D. Evolving object-oriented designs with refactorings. *Automated Software Engineering 8*, 1 (2001), 89–120. 1

[19] VAKILIAN, M., CHEN, N., NEGARA, S., RAJKUMAR, B. A., BAILEY, B. P., AND JOHNSON, R. E. Use, disuse, and misuse of automated refactorings. In *2012 34th International Conference on Software Engineering (ICSE)* (2012), pp. 233–243. 8

[20] VAKILIAN, M., AND JOHNSON, R. E. Alternate refactoring paths reveal usability problems. In *Proceedings of the 36th international conference on software engineering* (2014), pp. 1106–1116. 2, 4

[21] VERDUIN, E. *A Flexible Mechanism to Provide for a Refactoring Advice based on the Source Code Entity.* Open Universiteit, 2021. Master's Thesis. 7, 38, 49

[22] WERNSEN, W. *Source Code Query Systems for Detecting Refactoring Dangers.* Open Universiteit, 2022. Bachelor's Thesis. 7, 12, 47