Language Support for Refactorability Decay Prevention

Research Thesis

In Partial Fulfillment of the Requirements for the Degree of

Master of Science in Computer Science

Dov Fraivert

The Research Thesis Was Done Under

the Supervision of PROF. DAVID H. LORENZ

in the Dept. of Mathematics and Computer Science

The Open University of Israel

Submitted to the Senate of the Open University of Israel
September 2023

Acknowledgements

I would like to express my sincere appreciation and gratitude to my thesis advisor, Prof. David Lorenz, for the fruitful discussions and guidance throughout the process of writing this thesis. His assistance, attention to detail, and great ideas enriched my knowledge and made this thesis possible.

I would also like to thank Prof. Shachar Itzhaky for serving on my thesis defense committee and for his helpful comments and suggestions.

Abstract

Code smells are characteristics found in the code that indicate a violation of design principles that negatively impact the quality of the code. Even a code that is free of smells may be at high risk of forming them. In such cases, developers can either perform preventive refactoring in order to reduce the risk or leave the code as is and perform corrective refactoring as smells emerge. Each of these approaches has its advantages and disadvantages. On the one hand, developers usually avoid preventive refactoring during the development phase. This is because, at that point in time, the need is uncertain, and therefore the return on the investment is not guaranteed. On the other hand, when code smells eventually form, other developers who are less acquainted with the code, avoid the more complex corrective refactoring. As a result, a refactoring opportunity is missed, and the quality and maintainability of the code is compromised.

In this work, we treat refactoring not as a single atomic action, but rather as a sequence of subactions. This allows us to divide the responsibility for these subactions between the original developer of the code, who just prepares the code for refactoring, and a subsequent developer, who may need to carry out the actual refactoring action (when code smells form). To manage this division of responsibility, we have designed and developed a set of annotations along with an annotation processor that prevents software erosion from compromising the ability to perform the refactoring action.

List of Publications

[1] D. Fraivert and D. H. Lorenz. 2022. Language Support for Refactorability Decay Prevention. In *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '22)*, December 6–7, 2022, Auckland, New Zealand, 122-–134. ACM, New York, NY, USA. https://doi.org/10.1145/3564719.3568688

[2] D. Fraivert and D. H. Lorenz. 2022. Explicit Code Reuse Recommendation. In *Companion Proceedings of the ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH '22)*, December 5-11, 2022, Auckland, New Zealand, 9–10. ACM, New York, NY, USA. https://doi.org/10.1145/3563768.3564118

Contents

List of Figures x

List of Listings xi

List of Tables xiii

List of Algorithms xiv

List of Definitions xvi

1 Introduction 1

1.1 When to Refactor? 2

1.2 The Problem 3

1.3 Our Approach 4

1.3.1 Refactoring the Refactoring Action 4

1.3.2 Approach to Preventing Refactorability Decay 4

2 Background 7

2.1 Preventive Refactoring 7

2.1.1 Advantages 7

2.1.2 Disadvantages Due to Incompatibility 8

2.1.3 Disadvantages Due to Lack of Worthwhileness 9

2.2 Corrective Refactoring 9

2.2.1 Advantages 10

2.2.2 Disadvantages 10

2.3 Tools that Help Perform Corrective Refactoring 10

2.3.1 Identifying Refactoring Opportunities 10

2.3.1.1 Limitations: 11
2.3.2 Documentation 11
2.3.3 Tools Support for Refactoring 11
2.4 Advantages of the Our Approach 11

3 Problem **13**
3.1 Refactorability Decay Due to Lack of Familiarity With the Code 13
3.2 Refactorability Decay Due to Code Erosion 19
3.3 The Consequence of Refactorability Decay 19
3.4 Could Language Support have Helped to Prevent Refactorability Decay? . 25

4 Approach **29**
4.1 Extract Method 29
4.1.1 Sequence of Steps 29
4.1.2 Derivation of a Check-List of Refactorability Scents 34
4.2 Move Method 35
4.2.1 Sequence of Steps 35
4.2.2 Derivation of a Check-List of Refactorability Scents 37
4.3 Extract Method + Move Method 38

5 Language Support **39**
5.1 Properties of a Code Fragment 39
5.2 New Annotations 39
5.2.1 The `@MovableMethod` Annotation 40
5.2.2 The `@ExtractableCode` Annotation 40
5.2.3 The `@MovableCode` Annotation 41
5.2.4 Configurable Annotations 42
5.3 Implementation Notes 42

6 Methodology **45**
6.1 Movable Code 45
6.2 Movable Method 49
6.3 Annotation of Code that is Already Ready for Refraction 52

6.4 Code that is Difficult to Refactor . . . 53

7 Evaluation **55**

7.1 Simulating Developer δ_1 . . . 55

Experiment: . . . 56

7.2 Simulating Developer δ_3 . . . 57

Experiment: . . . 57

7.3 Results . . . 57

7.4 Threats to Validity . . . 59

8 Discussion **61**

8.1 Another Possible Benefit of Using Annotations . . . 61

8.2 Directions for Further Research . . . 63

8.2.1 Mapping the Entire Refactoring Catalog . . . 63

8.2.2 Annotation for Self-contained Code that is Suitable for Reuse . . . 64

8.2.3 Possible Improvements to the Annotations Presented in Sect. 5.2 . . . 64

8.2.4 Support for Refactorability Decay Prevention in Additional Languages 65

8.2.5 Semantic Changes in Reusable Code . . . 65

9 Conclusion **67**

A Developer Guide **69**

A.1 Overview of the Class Structure . . . 69

A.1.1 The `annotations` Package . . . 69

A.1.2 The `processors` Package . . . 69

A.1.3 The `engine` Package . . . 73

A.1.4 The `visitors` Package . . . 74

A.2 Guidelines for Adding New Annotations and Tests . . . 75

B User Guide **77**

B.1 Adding the COREAN Library to INTELLIJ . . . 77

B.2 Using Configurable Annotations . . . 78

B.3 Limitations . . . 79

B.4 The ReusableCodeViewer Plugin 80
B.4.1 Adding the ReusableCodeViewer Plugin to IntelliJ 80
B.4.2 Limitations 80

C Supplemental Examples 81

D Evaluation Tasks 89

D.1 Examples of the Tasks 89
D.2 Details of the Results 95

List of Figures

1.1 Example of the expected level of code familiarity and code erosion throughout the life of the project. 2
1.2 Example of the expected amount of preventive and corrective refactoring at three phases in the life of a project. 3

3.1 Attempt to perform EM via the refactoring menu. 20

8.1 The option Show Reusable Code added to Tools by the plugin 62

A.1 The class diagram of Corean 70

Listings

3.1 The method openWindow » JabRefGUI 14
3.2 The extracted method 15
3.3 The corrections that allow to perform MM 17
3.4 A method with a useful code fragment 18
3.5 The method BibtexCaseChanger » convertSpecialChar 20
3.6 The method FieldWriter » checkBraces 21
3.7 The method BibtexNameFormatter » formatName 22
3.8 The method BibtexNameFormatter » numberOfChars 23
3.9 The method BracesCorrector » apply 23
3.10 The method RemoveBracesFormatter » hasNegativeBraceCount 24
3.11 List. 3.6 with the @ExtractableCode annotation added 25
3.12 Build output for List. 3.11 26
3.13 List. 3.11 after possible modifications by developer δ_1 at time τ_1 27
3.14 Error message after annotating List. 3.5 with @ExtractableCode 27
3.15 Error message after annotating List. 3.10 with @ExtractableCode 27
4.1 Before extraction 31
4.2 After extraction 31
4.3 Before extraction 31
4.4 After extraction 31
4.5 Before extraction 32
4.6 After extraction 32
4.7 Before extraction 33
4.8 After extraction 33
6.1 The useful code annotated with @MovableCode 47
6.2 Error message caused by @MovableCode annotation 47

6.3 List. C.4 after changes by developer δ_2 . . . 48

6.4 Error message caused by `@MovableCode` annotation . . . 48

6.5 A useful **private** method . . . 50

6.6 Error message after annotating List. 6.5 with `@MovableMethod` . . . 51

6.7 Error message caused by the changes made by developer δ_2 . . . 51

6.8 List. 3.9 with the `@MovableCode` annotation added . . . 52

6.9 List. 3.5 with the `@ExtractableCode` annotation added . . . 53

B.1 The dependency of COREAN library . . . 78

B.2 The *refactorability_configuration.json* configuration file . . . 78

C.1 List. 3.1 after possible corrections of developer δ_1 . . . 82

C.2 The result of the extraction with ECLIPSE . . . 83

C.3 The result of the extraction with INTELLIJ . . . 83

C.4 List. 6.1 after the preparation step for EM and MM refactoring . . . 84

C.5 List. 6.3 after possible modifications by developer δ_2 that do not cause refactorability decay . . . 85

C.6 List. 6.5 after the preparation step for MM refactoring . . . 86

C.7 List. C.6 after the changes made by developer δ_2 . . . 87

C.8 List. C.7 after possible modifications by developer δ_2 that do not cause refactorability decay . . . 88

D.1 Refactoring Task A_i (without annotations) . . . 90

D.2 Refactoring Task A_i^* (with annotations) . . . 91

D.3 Refactoring Task B_i (without annotations) . . . 92

D.4 Refactoring Task B_i^* (with annotations) . . . 93

List of Tables

4.1 Refactorability Scents of EM 34
4.2 Refactorability Scents of MM 37

D.1 Comparison of challenges 94

List of Algorithms

1 *Extract Method* 30
2 *Move Method* 35
3 *Pull Up Method* 63

List of Definitions

4.1.1 Definition (EM-ready) 34
4.1.2 Definition (preparatory stage for EM) 34
4.2.1 Definition (MM-ready) 37
4.2.2 Definition (preparatory stage for MM) 37

Chapter 1

Introduction

Code refactoring [3, 4] is the process of applying a series of refactoring actions in response to code smells. A *refactoring action* [5] is a small, behavior-preserving code transformation, in which the structure of the code is changed without affecting its observable behavior. A *code smell* [5] is an anti-pattern found in the code that indicates a potential flaw which a refactoring action may correct.

One use of refactoring is as a preventive measure for making the code more robust when it is still free of smells [6]. Another more common use of refactoring [4] is for correcting *code smells* as they occur. We refer to the former use as *preventive refactoring* and to the latter as *corrective refactoring*.

The difference between "preventive" and "corrective" refactoring is not only in the time of initiation but also in the complexity and cost of achieving the same objective. *Preventive refactoring* is typically carried out by the developer when the code is still fresh and tidy. In contrast, *corrective refactoring* is typically done by some other developer who might be less familiar with the code [7], after erosion already took place [8]. We refer to the increasing complexity of code refactoring over time as *refactorability decay* (Fig. 1.1).

This begs the question: in the face of refactorability decay, which practice is more cost-effective — multiple *preventive refactorings* up-front when the cost of applying the refactoring is low but the return-on-investment is not guaranteed — or, fewer *corrective refactorings* down-the-road when the need is certain but the cost of applying the refactoring is higher?

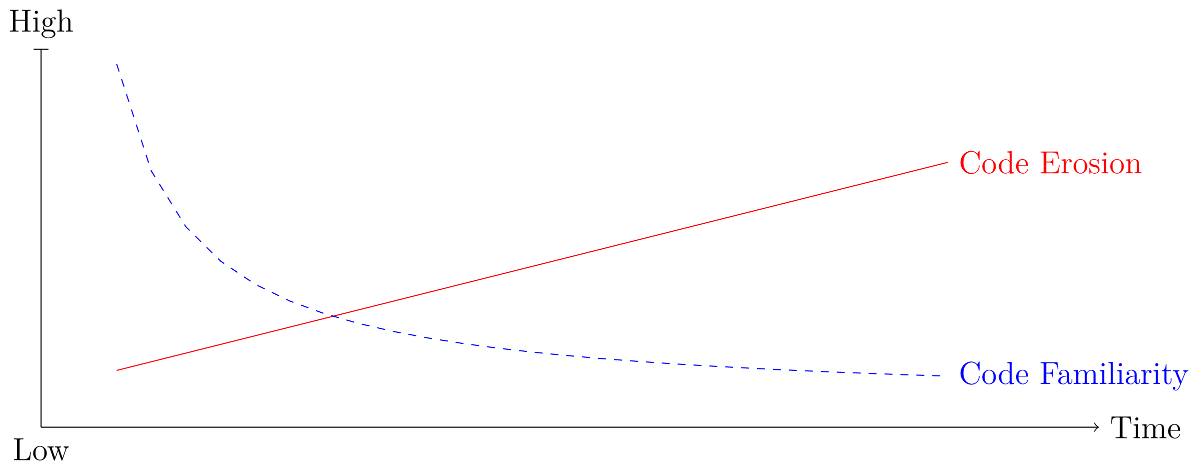

Figure 1.1: Example of the expected level of code familiarity and code erosion throughout the life of the project.

1.1 When to Refactor?

Timing a refactoring action is not a simple matter [9]. For example, consider code written by developer δ_1 at time τ_1 (Fig. 1.2). Then, at time τ_2 the code is maintained by another developer δ_2 (or several developers). Finally, at time τ_3 a third developer δ_3 improves or reuses the code. The three moments, τ_1, τ_2, and τ_3 may be close together or may be spread over an extended period of time. W.l.o.g., we can assume that at time τ_1 the code is free of smells (otherwise, developer δ_1 would have eliminated them with *corrective refactoring*).

Now, suppose that developer δ_1 foresees already at time τ_1 the likelihood for certain *code smells* forming and even has a good grasp on what sort of restructuring might be helpful in the long term. Still, there might be no urgency or resources to engage at time τ_1 in extensive *preventive refactoring*. There might also be concern that premature refactoring at time τ_1 can unnecessarily clutter the code, thus causing in the short term deterioration rather than improvement of the software quality.

In these cases as well as others, developer δ_1 would need to defer to developer δ_3 the appropriate *corrective refactoring* action, to be done at time τ_3 should *code smells* actually form. Unfortunately, there is neither channel for communicating information from developer δ_1 to developer δ_3 at time τ_1 or at time τ_3, nor language support for reducing the risk of *refactorability decay* between time τ_1 and time τ_3.

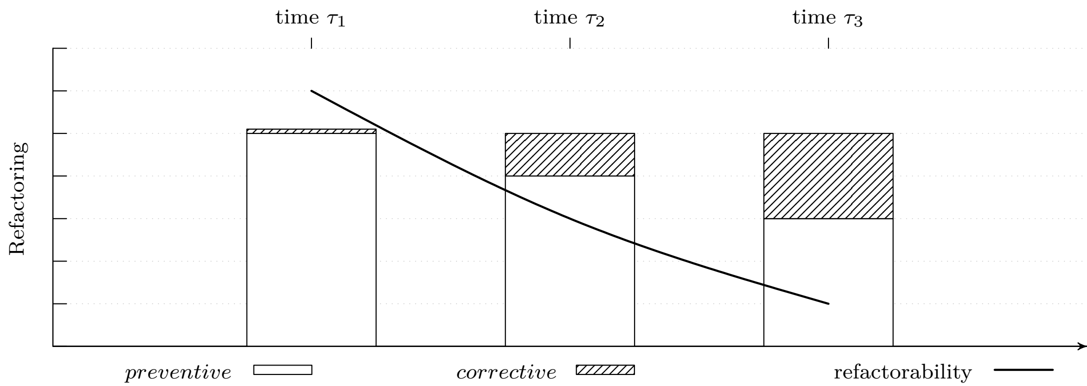

Figure 1.2: Example of the expected amount of preventive and corrective refactoring at three phases in the life of a project.

1.2 The Problem

As a result of the trade-offs that *preventive* and *corrective refactoring* present, in many projects developer δ_1 does not perform *preventive refactoring* at time τ_1, and when, in the future, *code smells* are formed, developer δ_3, who is not well-acquainted with the code, avoids performing complex *corrective refactoring* (e.g., because of the risk of introducing bugs into the existing code [10]), and as a result, the quality and maintainability of the code are compromised.

This problem is caused by all the required effort being concentrated in the same stage, either time τ_1 (for *preventive refactoring*) or time τ_3 (for *corrective refactoring*). We are looking for a new approach that will allow the effort to be distributed throughout the life of the project, between time τ_1, time τ_2, and time τ_3.

Our thesis is that if we identify the properties of the code that make *corrective refactoring* complex to execute for developer δ_3, who is not familiar with the code, then we can develop a method by which:

- Developer δ_1, instead of performing the full *preventive refactoring*, will only perform "code refactoring preparation", in which he will ensure that the code does not contain problematic properties. This requires significantly less effort than full *preventive refactoring*, so δ_1 developers will be more willing to perform it.
- Developer δ_3 will perform *corrective refactoring* on code that has passed the "preparation stage" (and therefore does not have the properties that can make it difficult to perform *corrective refactoring*). This refactoring will require significantly

less effort than *corrective refactoring* on regular code, so δ_3 developers will be more willing to perform the required refactoring.

1.3 Our Approach

In this work, we introduce a new approach to refactoring that strikes a balance between *preventive* and *corrective refactoring*.

1.3.1 Refactoring the Refactoring Action

The dilemma is whether to refactor proactively at time τ_1 or reactively at time τ_3 is based on a common but false perception that refactoring is an atomic action. However, a refactoring action is anything but atomic. Even with IDE support, each action encompasses a sequence of steps taken by the developer before and after accessing the "one-click" refactoring action through the context menu.

In this work, we identify these steps and show that they can be partitioned into two sets: σ-steps and μ-steps. A *σ-step* is structural in nature — it relies on the coding style and other properties, requires a deep understanding of the code, and is subject to refactorability decay. A *μ-step* is mechanical in nature — it does not require a deep understanding of the code and is less sensitive to code erosion.

We derive a check-list of "refactorability scents" that developer δ_1 should address at time τ_1, as preparation for the refactoring action. With this "preparatory step" and assuming that code erosion does not admit these scents back into the code at time τ_2, we can expect that developer δ_3 will be able to cope successfully with σ-steps at time τ_3, just as well as developer δ_1 could have coped with them at time τ_1. As for μ-steps, there is not much difference between developer δ_1 and developer δ_3 in term of the proficiency and effort required to complete them.

1.3.2 Approach to Preventing Refactorability Decay

If the refactoring action consists mainly of σ-steps, it is better that developer δ_1 performs the operation up-front. If the refactoring action consists mainly of μ-steps, it is best to

defer the refactoring to developer δ_3.

In case the refactoring operation comprises multiple σ-steps and multiple μ-steps, the responsibility can be shared. Developer δ_1 can make necessary changes to the code to eliminate refactorability scents, thus facilitating easy execution of σ-steps later on. Developer δ_1 can also annotate (annotations are presented in Chapter 5) code fragments relevant to the action, allowing the compiler to flag refactorability scents that could interfere with σ-steps. With these annotations in place, developer δ_2 would receive at time τ_2 compilation error messages alerting against undesirable changes, thus protecting these fragments of code against refactorability decay. Eventually, when developer δ_3 needs to perform the refactoring action, the lack of familiarity with the code should not play a critical role in the safe and easy completion of the σ-steps.

To concretely illustrate our approach, we present such annotations for the JAVA programming language. With these annotations, developer δ_1 can annotate code fragments. In turn, these annotations enable compile-time checks that help to reduce and hopefully prevent refactorability decay.

Chapter 2

Background

This chapter describes the background of the types of refactoring, the advantages and disadvantages of each type, and the tools that can help perform the refactoring.

2.1 Preventive Refactoring

There are good reasons for developer δ_1 to perform *preventive refactoring* at time τ_1. For example:

- A useful code fragment was written as part of a method in some class, and not as a separate method in an appropriate class. If in the future, this capability is needed elsewhere in the project, there is a risk that code duplication will be created.
- A method that is responsible for several actions: there is a risk that in the future this method will grow beyond the desired size.
- A **private** method that performs an operation unrelated to the role of the class: there is a risk that, in the future, this will cause *Feature Envy* [11].

In these cases, the developer δ_1 can perform *preventive refactoring* at time τ_1, and thus decrease the risk of the formation of *code smells* in the future.

2.1.1 Advantages

The refactoring will be done by the developer δ_1 who knows the code well (compared to developer δ_3), so a smaller investment of time and effort will be required, and there is a

lower risk that errors will be generated due to the refactoring.

Another advantage of performing refactoring at time τ_1 is that there is a risk that at time τ_2, *software erosion* will occur which will make it difficult to perform refactoring at time τ_3.

Concerning the three types of *code smells* that we mentioned at the beginning of the section, performing *preventive refactoring* at time τ_1 means ensuring *Single Responsibility* [12] at the level of methods and classes, as well as ensuring low *coupling* and high *cohesion* in developed classes. Such refactoring will produce code consisting of short methods that perform defined operations, and small classes that perform a defined role, and that are dependent mainly on their internal components. Alongside the advantages, such development has its drawbacks, due to incompatibility and lack of worthwhileness.

2.1.2 Disadvantages Due to Incompatibility

In the following cases, developer δ_1 can decide that it is less appropriate for refactoring at time τ_1:

- There are cases in which we prefer that the code remains within a single large method, for example, in order to see the full picture and improve efficiency in the future, so we would prefer not to perform *Extract Method* [13].

- Agile focuses on coding for current needs and does not encourage investment in coding that may be used for future reuse. For example, in *Extreme Programming*, we want to get the *Simplest Design*, and therefore we create a minimum number of classes and methods, without strict adherence to *Single Responsibility*, and only when in practice we want to reuse the existing code we do perform an appropriate refactoring operation [14].

- Some developers prefer a development style of writing long methods that perform several actions.

2.1.3 Disadvantages Due to Lack of Worthwhileness

Execution of *Preventive refactoring* whenever developer δ_1 detects any violation of the *Single Responsibility* principle requires a large investment at time τ_1, which is not always worthwhile to perform in that phase of initial development, since only a small part of the developed code can actually be reused in the future, or *code smells* will be created in it, which doesn't justify a great investment in the initial development phase [10, 15].

Another disadvantage arises from the fact that in the initial phase of the project, not all classes exist yet, so it can be difficult to find the most suitable class for some particular code. At this initial stage, if the developer of a particular class needs code that performs a certain action that is not related to the main role of the current class, sometimes it is more correct and effective to leave the code in the current class for the time being, and look for the most appropriate class for it only later in the project, when all the other classes already exist.

2.2 Corrective Refactoring

There are good reasons for developer δ_3 to perform *corrective refactoring* at time τ_3.

Even if the code is written without *code smells* at time τ_1, at time τ_2 the code undergoes changes and adjustments which can cause *software erosion* and formation of *code smells* in the code. For example:

- Code duplication can be created.
- Additional code that performs another action can be added to code that performed a particular action.
- A code segment can increasingly use other class elements.
- A method can grow beyond the desired size.

All of these problems must be addressed at time τ_3 by developer δ_3 who has detected the existence of previously created *code smells*, or who himself wants to make a change in the code that will cause *code smells* (e.g., inserting a fragment of code that already exists). In

this case, he must perform *corrective refactoring* appropriately and prevent the creation of *code smells*.

2.2.1 Advantages

The main advantage is that the refactoring will be performed only in those cases where the need is certain. For example, where in practice we want to reuse this code, or if other *code smells* are formed.

Another advantage comes from the fact that usually, time τ_3 is at a more advanced stage in a project when there are already more classes, and therefore, it is easier to find the class that best fits a particular code.

2.2.2 Disadvantages

The refactoring will be performed by developer δ_3 that did not develop the original code and does not necessarily know it thoroughly. Therefore, refactoring will require a greater investment of time and effort, and there is a higher risk of errors.

Another disadvantage comes from the risk that maintenance changes made at time τ_2 can generate dependence on, and a closer connection to, other code. This will make it difficult to perform the desired refactoring operation.

2.3 Tools that Help Perform Corrective Refactoring

2.3.1 Identifying Refactoring Opportunities

JDeodorant [16, 17] is a plugin for Eclipse that automatically identifies places in the code where two types of *Extract Method* can be performed: (i) a complete computation of a given variable; (ii) determining the state of a given object. This plugin also identifies methods for which *Move Method* to a particular target class will reduce *coupling* and improve *cohesion*. The algorithm screens the possible suggestions, and leaves only those in which the preservation of behavior is guaranteed. They report that about 50% of cases in which *Extract Method* could be performed were dismissed because it was not possible to guarantee the preservation of behavior.

2.3.1.1 Limitations:

There are two key limitations:

- This approach does not deal with cases in which the execution of refactoring would have improved the code, but the preservation of behavior is not guaranteed.
- This approach does not deal with cases in which refactoring seems correct to developers but does not improve the software metrics.

2.3.2 Documentation

Good documentation can help identify places where refactoring can be performed.

But it is not guaranteed that the code is written in a style that allows for easy refactoring. In addition, there is a risk that standard documentation can become out of date, due to changes made to the code at time τ_2

2.3.3 Tools Support for Refactoring

A code fragment selected for refactoring must be a list of valid statements. For instance, it must include the jump target for **continue** or **break** (if they exist) and a **return** from any possible path (if a **return** exists). *Selection Assist* [18] and *Box View* [18] are two tools that help select code fragments that are valid. A third tool, *Refactoring Annotations* [18], visualizes locations in the code with **continue**, **break** and **return** that should be manually corrected.

These tools can indeed help with refactoring actions that require manual intervention. However, these tools help with the technical aspect of the action, but they do not guarantee preservation of behavior or attempt to prevent refactorability decay.

2.4 Advantages of the Our Approach

The existing approaches do not give a complete answer to the question of when and how it is better to perform the refactoring. Compared to the previous approaches, the new approach we will present deals better with the limitations of the other approaches.

- It reduces the development effort of the developer δ_1 compared to *preventive refactoring*.
- Ensures that the code is written in a style that will facilitate easy execution of *corrective refactoring*, even by developers δ_3 who are not familiar with the code.
- It is appropriate for additional development methods.
- It protects against refactorability decay during the life of the project.
- Can deal with cases in which the refactoring action changes the behavior of the code (the necessary changes will be made by the developer δ_1).
- It identifies opportunities for refactoring that seems appropriate to developer δ_1 and is not limited to improving particular software metrics.

Chapter 3

Problem

The difficulty of applying refactoring actions varies depending on the state of the code. As a result, it may be necessary to perform a manual operation prior to refactoring, which may require a detailed understanding of the code. We will present several examples, most of them from real open-source projects, that will illustrate the problems in the current situation.

3.1 Refactorability Decay Due to Lack of Familiarity With the Code

As a motivation example, let us consider refactorability decay found in the JabRef project.[1] This example shows a code fragment that can be useful in other places in the project, but if we want actually to reuse this code, we will have to perform preliminary refactoring actions. This code is written in a style that does not make it easy to refactor using automated tools, and requires a preliminary manual change to the code.

In List. 3.1 the highlighted code defines the size of the window. This code can be useful but it is part of a long method. As a result, if at time τ_3, we want to reuse this code, we will have to perform EM and MM.

If we choose the highlighted segment and use the built-in refactoring tool of IntelliJ, we will get the result described in List. 3.2 (we manually changed the name).

[1] JabRef is an open-source, cross-platform citation and reference management tool (https://github.com/JabRef/jabref/).

Listing 3.1: The method openWindow » JabRefGUI

```
1  public class JabRefGUI {
2    ⋮ ~~~The beginning of the code omitted~~~
3    private boolean correctedWindowPos;
4    ⋮ ~~~Part of the code omitted~~~
5    private void openWindow(Stage mainStage){
6      mainFrame.init();
7      GuiPreferences guiPreferences = preferencesService.getGuiPreferences();
8      // Restore window location and/or maximized state
9      if (guiPreferences.isWindowMaximized()) {
10       mainStage.setMaximized(true);
11     } else if ((Screen.getScreens().size()==1)&&isWindowPositionOutOfBounds()){
12         // corrects the Window, if it is outside the mainscreen
13         mainStage.setX(0);
14         mainStage.setY(0);
15         mainStage.setWidth(1024);
16         mainStage.setHeight(768);
17         correctedWindowPos = true;
18     } else {
19         mainStage.setX(guiPreferences.getPositionX());
20         mainStage.setY(guiPreferences.getPositionY());
21         mainStage.setWidth(guiPreferences.getSizeX());
22         mainStage.setHeight(guiPreferences.getSizeY());
23     }
24   ⋮ ~~~The rest of the code omitted~~~
25   }
26 }
```

Listing 3.2: The extracted method

```
1  public class JabRefGUI {
2    ⋮ ~~~The beginning of the code omitted~~~
3    private boolean correctedWindowPos;
4    ⋮ ~~~Part of the code omitted~~~
5    private boolean setWindowSize(Stage mainStage){
6      GuiPreferences guiPreferences = preferencesService.getGuiPreferences();
7      // Restore window location and/or maximized state
8      if (guiPreferences.isWindowMaximized()) {
9          mainStage.setMaximized(true);
10     } else if ((Screen.getScreens().size() == 1)&&isWindowPositionOutOfBounds()){
11           // corrects the Window, if it is outside the mainscreen
12           mainStage.setX(0);
13           mainStage.setY(0);
14           mainStage.setWidth(1024);
15           mainStage.setHeight(768);
16           correctedWindowPos = true;
17     } else {
18           mainStage.setX(guiPreferences.getPositionX());
19           mainStage.setY(guiPreferences.getPositionY());
20           mainStage.setWidth(guiPreferences.getSizeX());
21           mainStage.setHeight(guiPreferences.getSizeY());
22     }
23   }
24
25   private void openWindow(Stage mainStage){
26         mainFrame.init();
27         setWindowSize(mainStage);
28         ⋮ ~~~The rest of the code omitted~~~
29   }
30 }
```

Next, to enable reuse, if we try to move the extracted method to a more general class using the automated refactoring tool of IntelliJ, we will receive an error message: "*Field `correctedWindowPos` is private and will not be accessible from method `setWindowSize(Stage)`.*"

The problem stems from the fact that the code assigns a value to the **private** instance variable `correctedWindowPos`, that the target class has no access to. In order to correct this problem, we must manually change the signature of the method that we extracted, and replace the assignment to `correctedWindowPos` with a return of the appropriate value. Here we encounter another problem, since not all flows in the original code assign a value to `correctedWindowPos`, and as a result, the new method will not return **boolean** for all the flows.

This problem cannot be easily solved even with automatic refactoring tools. In order to enable performing the MM on the extracted method, we must make another manual change that will ensure a return value in all flows, (for example, List. 3.3), and the code of the original method will be as in the lower part of List. 3.3.

Does the change we made preserve the behavior of the original code? Not necessarily, since now we assign a **false** value to `correctedWindowPos` in cases where in the original code we would not have assigned anything (and have left the previous value). In order to know whether this change affects the behavior or not, we must follow the flows of the original code, and make sure that the value of `correctedWindowPos` was always **false** before calling the `openWindow` method. This inspection may require an investment of time.

We will note that if we had asked this question of developer δ_1 of the original code, in most cases he would know the answer immediately, and as a result, he could write the original code in a different style, for example, in each flow assign a value to a temporary variable, and at the end assign the value of the temporary variable to `correctedWindowPos`, something that would have made it easy for us to perform EM and MM (List. C.1 on page 82).

Listing 3.3: The corrections that allow to perform MM

```
public class JabRefGUI {
    ⋮ ~~~The beginning of the code omitted~~~
    private boolean correctedWindowPos;
    ⋮ ~~~Part of the code omitted~~~
    private boolean setWindowSize(Stage mainStage){
      GuiPreferences guiPreferences = preferencesService.getGuiPreferences();
      // Restore window location and/or maximized state
      if (guiPreferences.isWindowMaximized()) {
          mainStage.setMaximized(true);
          return false;
      } else if ((Screen.getScreens().size()==1)&&isWindowPositionOutOfBounds()){
          // corrects the Window, if it is outside the mainscreen
          mainStage.setX(0);
          mainStage.setY(0);
          mainStage.setWidth(1024);
          mainStage.setHeight(768);
          return true;
      } else {
          mainStage.setX(guiPreferences.getPositionX());
          mainStage.setY(guiPreferences.getPositionY());
          mainStage.setWidth(guiPreferences.getSizeX());
          mainStage.setHeight(guiPreferences.getSizeY());
          return false;
      }
    }

    private void openWindow(Stage mainStage){
          mainFrame.init();
          correctedWindowPos = setWindowSize(mainStage);
          ⋮ ~~~The rest of the code omitted~~~
    }
}
```

Listing 3.4: A method with a useful code fragment

```java
public class Source {
    ⋮ ~~~The beginning of the code omitted~~~
    private String results;
    ⋮ ~~~Part of the code omitted~~~
    private void foo(String[] commandsArray, String[] paramsArray) {
        if (commandsArray.length == paramsArray.length) {
          int numOfErrors = 0; int numOfWarnings = 0;
          for (int i=0;i<commandsArray.length;i++) {
                if ((commandsArray[i].matches(".*[{`%].*")) ||
                    ↪ (paramsArray[i].matches(".*[{`%].*"))) continue;
                String p = paramsArray[i];
                String[] words = p.split("\\s+");
                if (words.length > 2) continue;
                String[] commands = {commandsArray[i], paramsArray[i]};
                Runtime rt = Runtime.getRuntime();
                Process proc = rt.exec(commands);
                BufferedReader stdInput = new BufferedReader(new
                    ↪ InputStreamReader(proc.getInputStream()));
                String outputLine = stdInput.readLine();
                while (outputLine != null) {
                  if (outputLine.contains("error")) numOfErrors++;
                  if (outputLine.contains("warning")) numOfWarnings++;
                  results += outputLine;
                  outputLine = stdInput.readLine();
                }
          }
          System.out.println("There were " + numOfErrors + " errors and " +
              ↪ numOfWarnings + " warnings");
        }
    }
    ⋮ ~~~The rest of the code omitted~~~
}
```

3.2 Refactorability Decay Due to Code Erosion

List. 3.4 illustrates a scenario that is typical in many projects. A method contains some useful fragments of code (lines 14-18, 21-23 highlighted in gray). This code accepts a command and its arguments, runs the command, and saves the output. Initially, at time τ_1, the method is written without the lines highlighted in yellow (lines 7, 19, 20, and 25). During the life of the project (at time τ_2) a subsequent developer adds the highlighted yellow lines (in our case, for counting the number of warnings and errors reported in the output of the command).

Suppose we need a similar functionality elsewhere in the project. In order to reuse the code, we must first find it. However, it is not obvious how to search for relevant code fragments. Even if the codebase is well-documented and somehow we manage to locate the desired functionality in the `foo` method, we still have to identify the relevant lines of code and extract them into a separate method in order to avoid code duplication [5]. In our case, the changes introduced in time τ_2 made the extraction difficult, because a method cannot return multiple values. The offending variables are `numOfErrors` and `numOfWarnings` that we do not necessarily need. We thus have to modify the code of the original method `foo`, and only then will we be able to extract the desired fragment into the separate method.

3.3 The Consequence of Refactorability Decay

Let us consider to additional refactorability decay found in the JabRef project. Various places in the project's code verify, for a given string, that the opening and closing braces are balanced (e.g., Listings 3.5 to 3.10). Surprisingly, however, the functionality is duplicated with small variations rather than refactored and reused. For example, the code in Listings 3.5 and 3.6 ignores escaped braces, while the code in Listings 3.7, 3.8, 3.9 and 3.10 counts also brace preceded with a '\'. As another example, the code in Listings 3.6 and 3.9 only checks that the total number of opening and closing braces in the entire string evens up, while the code in List. 3.5, List. 3.7, List. 3.8 and List. 3.10 checks that at no point in the string are there more closing braces than opening ones. Such duplicated functionality makes the code smelly, lengthy, and bulky, decreases its

Listing 3.5: The method BibtexCaseChanger » convertSpecialChar

```
1  int convertSpecialChar(StringBuilder sb, char[] c, int start, FORMAT_M format) {
2    ⋮ ∼∼∼The beginning of the code omitted∼∼∼
3    while ((i<c.length) && (braceLevel>0) && (c[i]!='\\')) {
4      if (c[i] == '}') {
5        braceLevel--;
6      } else if (c[i] == '{') {
7        braceLevel++;
8      }
9      i = convertNonControl(c, i, sb, format);
10   }
11   ⋮ ∼∼∼The rest of the code omitted∼∼∼
12 }
```

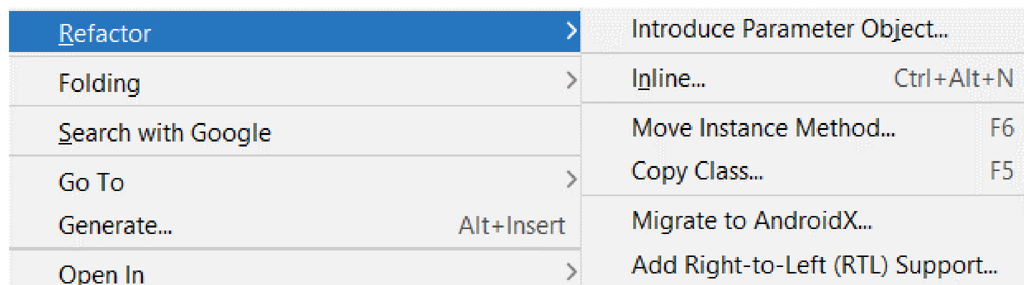

Figure 3.1: Attempt to perform EM via the refactoring menu.

quality, and increases technical debt.

In this example, it would be fair to assume that, whenever the developers needed such functionality, they preferred to reimplement it rather than refactor and reuse the code that already existed in the project. Indeed, although these code fragments seem simple, performing even basic refactoring actions on them is much harder than it first appears.

In List. 3.5 characters are converted within the loop by calling the `convertNonControl` method. If we want to extract the code that counts the brace into the `braceLevel` local variable, we would need to fully understand the action of `convertNonControl` and decide whether this conversion can be performed outside the **while** loop that counts the braces. If we choose the segment highlighted in List. 3.5 and try to use the built-in refactoring tool of IntelliJ, the possibility of extracting the code into a method would not even be presented (Fig. 3.1).

The reason for this is that we marked only part of the **while** statement and the automated tool does not know how to deal with such a situation.

In List. 3.6 the code counts right braces into the `right` local variable and left braces

Listing 3.6: The method FieldWriter » checkBraces

```java
private static void checkBraces(String text) {
  int left = 0; int right = 0;
  for (int i=0; i < text.length(); i++) {
    char item = text.charAt(i);
    boolean charBeforeIsEscape = false;
    if ((i>0) && (text.charAt(i-1) == '\\')) {
      charBeforeIsEscape = true;
    }
    if (!charBeforeIsEscape && (item=='{')) {
      left++;
    else if ((!charBeforeIsEscape) && (item == '}')) {
      right++;
    }
  }
  if (!(right == 0) && (left == 0)) {
    ⋮ ~~~Part of the code omitted~~~
  }
  if (!(right == 0) && (right < left)) {
    ⋮ ~~~Part of the code omitted~~~
  }
  if (left != right) {
    LOGGER.error("Braces don't match. Field value: {}", text);
    throw new InvalidFieldValueException("Braces don't match..."+text);
  }
}
```

into the `left` local variable. Further down the method, after the highlighted section, the values of these two variables are used. If we want to extract this fragment of code, we would encounter a new problem: the method extracted would need to return two values (for `right` and for `left`). In this case, if we choose the highlighted code segment in List. 3.6 and try to use the automated refactoring tool of IntelliJ, we would receive an error message: "*Unable to extract method. There are multiple variables to return.*" Therefore, before extraction, we would need to manually replace the use of the `right` and `left` variables with a single variable that stores the balance. As a result of this change, we would also need to update the three **if** statements at the end of the method.

In List. 3.7 a special check is performed inside the loop for handling the top level. Therefore, in order to extract the code that counts the braces, we would need to fully understand the surrounding code, and to decide whether or not the treatment can be

Listing 3.7: The method BibtexNameFormatter » formatName

```
1  static String formatName(Author author, String format, Warn warn) {
2    int level = 0; int i = 0;
3    ⋮ ~~~The beginning of the code omitted~~~
4    while ((i < n) && (level > 0)) {
5        wholeChar.append(c[i]);
6        if (c[i] == '{') {
7           level++;
8           i++;
9           continue;
10       }
11       if (c[i] == '}') {
12          level--;
13          i++;
14          continue;
15       }
16       if ((braceLevel==1) && Character.isLetter(c[i])) {
17         if ("fvlj".indexOf(c[i]) == -1) {
18            if (warn != null) {
19               warn.warn("Format string in...");
20            }
21         } else {
22             level1Chars.append(c[i]);
23         }
24       }
25       i++;
26   }
27   ⋮ ~~~The rest of the code omitted~~~
28 }
```

Listing 3.8: The method BibtexNameFormatter » numberOfChars

```
1  public static int numberOfChars(String token, int inStop) {
2      int result = 0; int braceLevel = 0;
3      ⋮ ~~~The beginning of the code omitted~~~
4      while ((i < n) && (braceLevel > 0)) {
5         if (c[i] == '}') {
6               braceLevel--;
7         else if (c[i] == '{') {
8               braceLevel++;
9         }
10        i++;
11     }
12     ⋮ ~~~The rest of the code omitted~~~
13 }
```

Listing 3.9: The method BracesCorrector » apply

```
1  public static String apply(String input) {
2    ⋮ ~~~The beginning of the code omitted~~~
3    String c = matcher.replaceAll("");
4    long diff = c.chars().filter(ch->ch=='{').count() -
         ↪ c.chars().filter(ch->ch=='}').count();
5    ⋮ ~~~The rest of the code omitted~~~
6  }
```

performed at the top level, separately from the loop that counts the braces. Similarly to List. 3.5, in this case it is not possible to use the automated refactoring tool, because only part of the **while** statement is marked.

In List. 3.8 the code updates the value of two variables (`braceLevel` and `i`) whose values are used later on. If we want to extract the code that counts the braces, we would need to return two values from the method. Once again, if we try to use the automated refactoring tool, we would receive the same error message as in List. 3.6.

In List. 3.9 the code that keeps track of the number of braces is separate from the rest of the code of the method, and therefore we can use the automated refactoring tool. In this case, if a need for this functionality arises elsewhere in the project, the challenge would be to find this particular fragment of code.

In List. 3.10 the `hasNegativeBraceCount` method only checks that there are no more closing than opening braces. The section highlighted saves the total number of braces into

Listing 3.10: The method RemoveBracesFormatter » hasNegativeBraceCount

```
1  private boolean hasNegativeBraceCount(String val) {
2    int braceCount = 0;
3    for (int index=0; index<val.length(); index++) {
4      char charAtIndex = val.charAt(index);
5      if (charAtIndex == '{') {
6        braceCount++;
7      } else if (charAtIndex == '}') {
8        braceCount--;
9      }
10     if (braceCount < 0) {
11       return true;
12     }
13   }
14   return false;
15 }
```

the braceCount variable, but this variable is not referenced outside the loop. Instead, the method returns a **boolean** value, whose value depends on the control flow. If we want to extract the fragment of code that counts the braces, we would first need to rewrite the code, while preserving its original behavior, e.g., replacing the **return** command inside the loop with a **break** command, and outside the loop returning the expression "(braceCount < 0)."

We note that if we try to extract the marked code fragment using the automated refactoring tool without manually rewriting the code first, it would technically be possible to do so. However, the result would not meet our needs. The automated refactoring tool of Eclipse generates a method that does not compile (List. C.2 on page 83). Whereas the automated refactoring tool of IntelliJ can handle a marked code segment that does not have a return value from all the flows by automatically adding **return false** at the end of the extracted method (List. C.3 on page 83). The generated method compiles but the code just duplicates the original method, rather than in a method that just returns the balance of braces.

Listing 3.11: List. 3.6 with the @ExtractableCode annotation added

```
@ExtractableCode(Description = "Calculate the balance between { and }")
private static void checkBraces(String text){
  /*@ExtractableBegin*/
  int left = 0; int right = 0;
  for (int i=0; i < text.length(); i++) {
    char item = text.charAt(i);
    boolean charBeforeIsEscape = false;
    if((i>0) && (text.charAt(i-1)=='\\')){
      charBeforeIsEscape = true;
    }
    if(!charBeforeIsEscape && (item=='{')){
      left++;
    } else if(!charBeforeIsEscape && (item == '}')) {
      right++;
    }
  }
  /*@ExtractableEnd*/
  if (!(right == 0) && (left == 0)) {
    ⋮ ~~~Part of the code omitted~~~
  }
  if (!(right == 0) && (right < left)) {
    ⋮ ~~~Part of the code omitted~~~
  }
  if (left != right) {
    LOGGER.error("Braces don't match. Field value: {}", text);
    throw new InvalidFieldValueException("Braces don't match..."+text);
  }
}
```

3.4 Could Language Support have Helped to Prevent Refactorability Decay?

We can only speculate why developer δ_1, who must have authored some of these code fragments, did not extract them into a separate method in a class that is accessible from everywhere in the project. Perhaps, developer δ_1 did not want to invest time in performing the extraction in practice. If this is the case, we can offer developer δ_1 a lightweight alternative.

List. 3.11 is the same as List. 3.6 with the relevant fragment of code marked for reuse with the @ExtractableCode annotation (which we shall introduce in Chapter 5). As a result, the annotation processor emits the error message displayed in List. 3.12, reflecting

Listing 3.12: Build output for List. 3.11

```
1 java: Code fragment cannot be extracted because there are multiple variables
    ↪ to return. A code fragment marked as extractable in method checkBraces
    ↪ of the class FieldWriter contains 2 variables (left and right) which
    ↪ change their value in the extractable fragment but are used outside
    ↪ this fragment. Multiple variables cannot be the return value of an
    ↪ extracted method.
```

the problems that need to be fixed before extraction of this code fragment into a method is possible.

Continuing this example, had the `@ExtractableCode` annotation been added to the code at time τ_1, then developer δ_1 would most likely have modified the code into, e.g., the code shown in List. 3.13, replacing the variables `left` and `right` with a single variable `balance`. Note that this seemingly harmless modification slightly changes the behavior of the original method — we merged three **if** statements into two. Developer δ_1 is in the best position to decide whether or not this change in behavior is acceptable. Once the code is corrected, however, it would be easy for developer δ_3 to complete the extraction of the code fragment marked in List. 3.13 at a later time with just a few mouse clicks by using INTELLIJ's automated refactoring tool.

Similarly, marking the fragment relevant to reuse in List. 3.5 would emit the compilation error message displayed in List. 3.14.

In contrast, marking the relevant fragment in List. 3.9 passes compilation without errors. And if we annotate the code fragment marked in List. 3.10, we would get the compilation error message displayed in List. 3.15.

In all these cases too, once the code is annotated and passes compilation, developer δ_3 should be able to apply the refactoring actions without difficulty.

Listing 3.13: List. 3.11 after possible modifications by developer δ_1 at time τ_1

```
1  @ExtractableCode(Description = "Calculate␣the␣balance␣between␣{␣and␣}")
2  private static void checkBraces(String text){
3    /*@ExtractableBegin*/
4    int balance = 0;
5    for (int i=0; i < text.length(); i++) {
6      char item = text.charAt(i);
7      boolean charBeforeIsEscape = false;
8      if((i>0) && (text.charAt(i-1)=='\\')){
9        charBeforeIsEscape = true;
10     }
11     if(!charBeforeIsEscape && (item=='{')){
12       balance++;
13     } else if(!charBeforeIsEscape && (item == '}')) {
14       balance--;
15     }
16   }
17   /*@ExtractableEnd*/
18   if (balance<0) {
19     ⋮ ~~~Part of the code omitted~~~
20   }
21   if (balance>0) {
22     ⋮ ~~~Part of the code omitted~~~
23   }
24 }
```

Listing 3.14: Error message after annotating List. 3.5 with @ExtractableCode

```
1  java: Code fragment cannot be extracted. A code fragment marked as extractable
     ↪ in method convertSpecialChar of the class BibtexCaseChanger contains a
     ↪ broken while loop statement.
```

Listing 3.15: Error message after annotating List. 3.10 with @ExtractableCode

```
1  java: Code fragment cannot be extracted. A code fragment marked as extractable
     ↪ in method hasNegativeBraceCount of the class JabRefGUI contains a
     ↪ return statement but there exists a path without a return statement.
```

Chapter 4

Approach

We illustrate our approach on the two most frequently used refactoring actions from Fowler's catalog [11]:

- *Extract Function* [11, p. 106]: modularizing a fragment of code as a function,[1] e.g., making a specific repeated fragment into its own new method – often referred to as *Extract Method* (EM) [5, p. 110];
- *Move Function* [11, p. 198]: grouping a function with related functions, e.g., moving a method to a more appropriate class – often referred to as *Move Method* (MM) [5, p. 142].

We break down each refactoring action into a sequence of steps in order to identify refactorability scents that can make the refactoring of the code difficult. At each step we assess the level of understanding of the code that is required to carry it out in order to classify the step as either σ-step (structural) or μ-step (mechanical, Sect. 1.3.1).

4.1 Extract Method

4.1.1 Sequence of Steps

Moving code out of method M_1 into a new method M_2 can be broken down into the seven steps listed in Alg. 1. Four of them are σ-steps and three are μ-steps. We explain each step and discuss the reasons for classifying it as either structural or mechanical.

[1] Inverse of *Inline Function* [11] expansion.

Algorithm 1 *Extract Method*

[EM-1_σ] Identify in M_1 the code to be extracted.

[EM-2_σ] Decontextualize the code to be extracted.

[EM-3_σ] Rationalize the code to be extracted.

[EM-4_σ] Methodize the code to be extracted.

[EM-5_μ] Name the new method M_2.

[EM-6_μ] Move the code out of M_1 and into M_2.

[EM-7_μ] Add to M_1 a call to M_2.

(1) The first step ([EM-1_σ]) is to *identify in the original method M_1 the code to be extracted*. However, it is not always clear where the relevant fragment begins and where it ends, and whether or not it includes unnecessary statements. Therefore, this step requires an understanding of the structure of the code, and we classify it as a σ-step.

Note that automated refactoring tools can identify physical fragments that can technically be extracted (e.g., computation of a given variable [17]), but they do not do a good job in identifying logical fragments that perform the task we are interested in. In order to identify these code fragments, an understanding of what the code is intended to do is needed.

(2) The second step ([EM-2_σ]) is to *decontextualize the code to be extracted*, i.e., separating the relevant from the irrelevant statements. However, there are situations in which the original behavior of the code is not necessarily preserved after the separation [17].

The code separated may contain a duplicate statement that affects the state of a shared object (i.e., code that is copied to the new method M_2 but also remains in the original method M_1). For example, in List. 4.2 after separating the code to be extracted from the code we want to leave in place (highlighted in List. 4.1), `lirt.hasNext()` is duplicated but still changes the state of the shared object `litr`. In this example, calling the `extractedMethod` method after extraction will exhaust the list. Consequently, the condition of the **while** loop remaining in the original method will always return **false**, and the code left inside the loop will never be executed.

The code separated may also contain a duplicated condition that checks the value of a

Listing 4.1: Before extraction

```
1 void someMethod(List<A> list) {
2   ListIterator<A> litr =
      ↪ list.listIterator();
3   while (litr.hasNext()) {
4     //Code we want to extract
5     //Code we want to leave.
6   }
7 }
```

Listing 4.2: After extraction

```
1 void someMethod(List<A> list) {
2   ListIterator<A> litr =
      ↪ list.listIterator();
3   extractedMethod (litr);
4   while (litr.hasNext()) {
5     //Code we have left.
6   }
7 }
8 void extractedMethod
    ↪ (ListIterator<A> l) {
9   while (l.hasNext()) {
10    //Code we have extracted.
11  }
12 }
```

Listings 4.1 and 4.2: A duplicate statement that changes the state of a shared object: before (List. 4.1) and after (List. 4.2) extraction.

Listing 4.3: Before extraction

```
1 public void paint(Window w) {
2   boolean scale = false;
3   this.width = w.getWidth();
4   this.height = w.getHeight();
5   this.scaleW = 1.0;
6   this.scaleH = 1.0;
7
8   if (this.width < this.MIN) {
9     this.scaleW = this.width / this.MIN;
10    this.width = this.MIN;
11    scale = true;
12  } else if (this.width > this.MAX) {
13    this.scaleW = this.width / this.MAX;
14    this.width = this.MAX;
15    scale = true;
16  }
17 }
```

Listing 4.4: After extraction

```
1 public void paint(Window w) {
2   calcWidth(w.getWidth());
3   boolean scale = false;
4   this.height = w.getHeight();
5   this.scaleW = 1.0;
6   this.scaleH = 1.0;
7   if (this.width < this.MIN) {
8     this.scaleW=this.width/this.MIN;
9     scale = true;
10  } else if (this.width > this.MAX) {
11    this.scaleW = this.width / this.MAX;
12    scale = true;
13  }
14 }
15 void calcWidth(double width) {
16   this.width = width;
17   if (this.width < this.MAX) {
18     this.width = this.MIN;
19   } else if (this.width > this.MAX) {
20     this.width = this.MAX;
21   }
22 }
```

Listings 4.3 and 4.4: A condition that checks the value of a variable: before (List. 4.3) and after (List. 4.4) extraction.

Listing 4.5: Before extraction

```
int fact = 1;
double e = 1;
int k = 10;
for (int i=1; i<k; i++) {
  fact = fact * i;
  e += 1/fact;
}
```

Listing 4.6: After extraction

```
double e = 1;
int k = 10;
int fact = calcFact(k);
for (int i=1; i<k; i++) {
  e += 1/fact;
}
int calcFact(int k) {
  int fact = 1;
  for (int i=1; i<k;i++) {
    fact = fact * i;
  }
  return fact;
}
```

Listings 4.5 and 4.6: Using an intermediate value: before (List. 4.5) and after (List. 4.6) extraction.

variable. For example, the code in List. 4.3 calculates and assigns a value to the `scaleW`, `width`, and `scale` instance variables. After extracting the code that calculates the value of `width` to a separate method (the relevant code is highlighted in List. 4.3), the condition in the extracted code is duplicated in the remaining code. Since the extracted method, `calcWidth`, changes the value of the `width` variable, which is inspected in the duplicated code, the condition in the remaining code will never be met.

It is also possible that the remaining code relies on intermediate values in the code separated. For example, the code in List. 4.5 calculates the value of $k!$, but also uses the intermediate values of the calculation to approximate $e \doteq \Sigma_{k=0}^{\infty} \frac{1}{k!}$. After extraction, the remaining code uses incorrectly the final value of $k!$ instead of the intermediate values.

Note that in all these cases, the difficulty in preserving behavior is due to the fact that the code we want to extract is intertwined with the code that remains in the original method. In case the code to be extracted is contiguous, there is no need for this step, and the task becomes much simpler. However, in general, unless we know that the relevant code is separate from the rest of the code of the method, this step may require deep understanding of the code and thus classified as structural.

(3) The third step ([EM-3_σ]) is to *rationalize the code*, i.e, fix syntax errors in the code to be extracted. After we have separated the code to be extracted from the rest of the code in the method, invalid code statements may have been generated, e.g., a dangling **else** statement without its **if**, or **continue** or **break** commands without a jump destination.

Listing 4.7: Before extraction

```
String[] cArr = {"cd", "C:\\"};
String[] pArr = {"mkdir", "tmp"};
Runtime rt=Runtime.getRuntime();
String[] first={cArr[0],pArr[0]};
rt.exec(first);
String[] sec={cArr[1],pArr[1]};
rt.exec(sec);
```

Listing 4.8: After extraction

```
String[] cArr = {"cd", "C:\\"};
String[] pArr = {"mkdir", "tmp"};
Runtime rt = getRuntime({cArr[0],
    ↪ pArr[0]});
String[] sec={cArr[1],pArr[1]};
rt.exec(sec);

Runtime getRuntime(String[] f) {
 Runtime rt=Runtime.getRuntime();
 String[] first = f;
 rt.exec(first);
}
```

Listings 4.7 and 4.8: Using the refactor tool of IntelliJ: before (List. 4.7) and after (List. 4.8) extraction.

In such cases, a manual adjustment to the code is required, which may require a deeper understanding of the code, thus we classify this step as a σ-step.

(4) The fourth step ([EM-4_σ]) is to *methodize the code to be extracted*, i.e., modify the code to be extracted so that it can become a method. A method may return only a single value, and if the code to be extracted contains a **return** command, it must **return** from all paths.

In languages that do not support in-out parameters, if the code we wish to extract assigns a value to a local variable that is used outside the code fragment, we must return the value assigned to that variable as the method's return value, and assign the return value to a local variable. In order to return this variable's value as a return value, it is necessary that a value is assigned to a single variable, and that the assignment is made from all possible paths. Code that does not meet these requirements needs to be manually modified, which requires a deep understanding of the code. We thus classify this step also as a σ-step.

(5) The fifth step ([EM-5_μ]) is to *name the new method M_2*. Selecting a good and meaningful name may be difficult, but this step does not require a deep understanding of the code structure, we thus classify it as mechanical.

(6) The sixth step ([EM-6_μ]) is to *perform the actual extract operation*. After the previous steps have been completed, this step is usually mechanical and supported by the IDE tool.

(7) The seventh step ([EM-7_μ]) is to *call the new method M_2 from the original method*

Table 4.1: Refactorability Scents of EM

Properties that can prevent refactoring	Stages that are liable to be affected
Noncontiguous list of statements	[EM-1_σ], [EM-2_σ]
Invalid list of statements	[EM-3_σ]
Continue or break without a jump destination	[EM-3_σ]
Return from some but not all paths	[EM-4_σ]
Writes to more than one local variable that is used outside the fragment	[EM-4_σ]

M_1. Care is needed in calling the new method with the appropriate arguments, and verifying that the code passes compilation. We can usually do this with an IDE tool, but sometimes we have to make manual changes. For example, the code in List. 4.7 runs two `cmd` commands in order to create the `C:\tmp` folder. If we use the automated refactoring tool of IntelliJ to perform the EM action on the marked section (which is a useful code fragment that receives and runs a `cmd` command), we get the code shown in List. 4.8. In this code, the marked line does not compile, because in Java we cannot initialize an array without declaring a new variable or creating a new object with **new**. In addition, we must return a `Runtime` value from the new method, or, alternatively, pass it as an additional argument.

Although such manual changes require an investment of time, their correct execution mainly requires knowledge of programming rather than familiarity with the code structure. Therefore, this step is classified as mechanical.

4.1.2 Derivation of a Check-List of Refactorability Scents

In the Sect. 4.1.1 we analyzed the stages that make up the EM refactoring action and the difficulties that can arise in the correct execution of each stage. Accordingly, it is possible to derive a list of properties that can cause difficulties in the performance of EM. Table 4.1 summarizes the problematic properties and the steps that can be affected.

Definition 4.1.1 (EM-ready) *A code fragment that does not contain characteristics from Table 4.1 is considered ready for EM.*

Definition 4.1.2 (preparatory stage for EM) *The preparatory stage for* EM *is an activity that ensures that the code fragment intended for extraction does not contain characteristics from Table 4.1.*

After execution of the preparatory stage, we can expect that EM can be performed (now or in the future) without dealing with the refactoring challenges that can require a deep understanding of the surrounding code.

4.2 Move Method

4.2.1 Sequence of Steps

Moving method M out of class C_1 into class C_2 can be broken down into the seven steps listed in Alg. 2. Three of them are σ-steps and four are μ-steps.

Algorithm 2 *Move Method*

[MM-1_μ] Find the most suitable target class C_2 for method M.

[MM-2_σ] Separate method M from the context of the source class C_1.

[MM-3_μ] Adjust calls to M.

[MM-4_σ] Verify that M's original behavior did not change.

[MM-5_μ] Move method M to the target class C_2.

[MM-6_μ] Replace calls to $C_1.M$ with calls to $C_2.M$.

[MM-7_σ] Verify that after moving method M to C_2 the original behavior of the code is preserved.

(1) The first step ([MM-1_μ]) is to *find the most suitable target class C_2 for method M.* There are automated tools that can help find the class to which a method transfer can reduce *coupling* and improve *cohesion* [16, 19], but if we want to find the most conceptually appropriate class, we will have to search for it manually. This step requires general familiarity with the entire project, but not deep understanding of the source class or the code of the method that we want to move, thus we classify this step as mechanical.

(2) The second step ([MM-2_σ]) is to *separate the method M from the context of the source class C_1.* This step raises several challenges.

First, when the method to be moved reads the value of an instance variable, we need to change the signature of the method and pass the variable as an argument. When the method writes to an instance variable, the situation is more complex. As long as just a single instance variable is used and the method returns **void**, we can return the new value as the return value of the method. However, if several instance variables are used, we have to pass them all to the method as in-out parameters, which is not always supported in the language.

Second, when the method to be moved calls another **public** method of the source class, we can still call it after the move (it only requires adjusting the code at the call site). However, when the method to be moved calls a **private** method, we have to either make that method **public** (not recommended in most cases) or move both methods together to the target class. To do this, we must make sure that the second method can also be easily moved to the new class, while preserving its behavior.

These challenges often require a change to the code structure, so we classify this step as structural.

(3) The third step ([MM-3_μ]) is to *adjust calls to M* to changes made to its signature in step [MM-2_σ]. Changes to the arguments or to the return value of M may be required. This step is mainly mechanical.

(4) The fourth step ([MM-4_σ]) is to *ensure that the changes made in steps* [MM-2_σ] *and* [MM-3_μ] *did not change the original behavior*. Changing the return value of M can change the behavior of the code that uses this method. Therefore, we classify this step as σ-step.

(5) The fifth step ([MM-5_μ]) is to actually *move the method M to the target class C_2*. This step is mechanical.

(6) The sixth step ([MM-6_μ]) is to *replace all calls to $C_1.M$ with calls to $C_2.M$*. This step is mechanical. In some cases, (especially when M is a **static** method), steps [MM-5_μ] and [MM-6_μ] can be completed with IDE tools.

(7) The seventh step ([MM-7_σ]) is to *verify that after moving method M to C_2 the original behavior of the code is preserved.*

There are several situations in which moving a method to another class can change the code behavior [20, 21]. In JAVA, if M is an implementation of a method declared **abstract**

Table 4.2: Refactorability Scents of MM

Properties that can prevent refactoring	Stages that are liable to be affected
Writes to an instance variable	[MM-2_σ], [MM-4_σ]
Locks on the current object	[MM-2_σ], [MM-7_σ]
Calls a **private** method	[MM-2_σ], [MM-4_σ]
Overrides a method	[EM-7_μ]
Makes the class **abstract**	[EM-7_μ]

in superclass, then after moving it to another class it will not be possible to instantiate the source class. In C++, if M was the only pure **virtual** function in the source class, then the class will no longer be considered **abstract**, and might be accidentally instantiated.

If M overrides a method defined in the superclass, moving M to another class might change the behavior of the subclass. If M is locked on the source class object (e.g., **synchronized** in Java), and another method in the source class locks on the same object, then after the move, M and the other method will now lock on different objects, and may unexpectedly run concurrently.

All of these cases require an understanding of the code, we thus classify this step as structural.

4.2.2 Derivation of a Check-List of Refactorability Scents

According to the analysis made in the Sect. 4.2.1, it is possible to derive a list of properties that can cause difficulties in performing MM. Table 4.2 summarizes the problematic properties and the steps that can be affected.

Definition 4.2.1 (MM-ready) *A method that does not contain characteristics from Table 4.2 is considered ready for MM.*

Definition 4.2.2 (preparatory stage for MM) *The preparatory stage for MM is an activity that ensures that the method intended for moving does not contain characteristics from Table 4.2.*

After execution of the preparatory stage, we can expect that MM can be performed (now or in the future) without dealing with the refactoring challenges that can require a deep understanding of the surrounding code.

4.3 Extract Method + Move Method

If we want to first extract a code fragment and then move it to another class, we need to follow the EM steps in Alg. 1 and then the MM steps in Alg. 2. However, with respect to preserving the behavior, it is sufficient to preserve the behavior observed before the EM action [22], thereby eliminating some of the problems mentioned in Sect. 4.2. For example, it would not be necessary to check before performing the MM action that the method is not **abstract** and does not override a method in the superclass.

Chapter 5

Language Support

5.1 Properties of a Code Fragment

Typically, compile-time verify that certain properties are present throughout the code. But one can think of unique tests that should run on only a certain method or only on a certain piece of code, while the rest of the code should remain with only the usual tests.

For example, consider a project written in Java-11, and contains a certain method that we want to include in the framework of this project, but in the future, we plan to reuse it in another project that is written in Java-5. In this case, we want to check that this method does not contain commands that are not included in Java-5. As another example, consider a project that has a certain method whose execution time is critical, while the execution time of the rest of the code is less important, we might want to check at compilation time that this method does not create new objects.

In this work, we would like to check during compilation that certain fragments of code and methods intended for refactoring do not have the characteristics that we defined in Sect. 4.1 and Sect. 4.2.

5.2 New Annotations

We implemented a library for the Java programming language, named *Collaborative Refactoring Annotations (Corean)*, for managing the sharing of responsibility for three refactoring actions: EM, MM, and EM+MM.

The library defines three annotations:

`@MovableMethod`, `@ExtractableCode`, and `@MovableCode`;

and two pairs of pseudo annotations:

/@ExtractableBegin*/* and */*@ExtractableEnd*/*, and

/@MovableBegin*/* and */*@MovableEnd*/*. It also provides an annotation processor that runs during compilation and reports refactorability scents.

Each annotation validates for a particular refactoring action that the annotated code does not have certain properties that are known to make σ-steps difficult to execute.

5.2.1 The `@MovableMethod` Annotation

Annotating a method with `@MovableMethod` indicates that we may want to apply the MM refactoring action. The annotation asserts the following conditions during compilation:

ϕ_I The method does not assign a value to an instance variable.

ϕ_{II} The method is not locked on the current object (i.e., **this**).

ϕ_{III} The method does not call a **private** method that is not marked with `@MovableMethod`.

ϕ_{IV} The method does not override a method of a superclass.

A method that satisfies these four properties facilitates the completion of steps [MM-2_σ], [MM-4_σ], and [MM-7_σ] even without deep understanding of the code.

5.2.2 The `@ExtractableCode` Annotation

Annotating a method with `@ExtractableCode` indicates that we may want to apply the EM refactoring action. The annotation consists of two parts:

- Annotating the method containing the code fragment with `@ExtractableCode`.
- Marking the code fragment that we may want to extract with */*@ExtractableBegin*/* at the beginning of the fragment and with */*@ExtractableEnd*/* at the end of the fragment.

This annotation ensures that:

ϕ_V The code fragment is contiguous (because, technically, only a contiguous fragment of code can be marked.)

In addition, the annotation asserts the following conditions during compilation:

ϕ_{VI} The code fragment contains only a valid and complete list of statements.

ϕ_{VII} If the code fragment contains **continue** or **break** commands, then it also contains the jump destination.

ϕ_{VIII} If the code fragment contains a **return** command, then a **return** exists in all paths.

ϕ_{IX} The code fragment contains an assignment to at most one local variable that is used after the marked fragment.

A code fragment that satisfies these five properties facilitates the completion of steps [EM-1_σ], [EM-2_σ], [EM-3_σ] and [EM-4_σ] even without a deep understanding of the code.

5.2.3 The `@MovableCode` Annotation

Annotating a method with `@MovableCode` indicates that we may want to apply the EM+MM refactoring action. The annotation consists of two parts:

- Annotating the method containing the code fragment with `@MovableCode`.
- Marking the code fragment on which we want to enable the EM and then MM to be performed with */*@MovableBegin*/* at the beginning of the fragment, and */*@MovableEnd*/* at the end of the fragment.

This annotation asserts that the marked code fragment satisfies the five conditions of `@ExtractableCode`. The annotation also assert the following additional conditions:

ϕ_X The marked fragment does not contain an assignment to an instance variable.

ϕ_{XI} The marked fragment is not locked on the current object (i.e., **this**).

ϕ_{XII} The code does not call a **private** method that is not annotated with `@MovableMethod`.

5.2.4 Configurable Annotations

We also include two configurable annotations, `@MethodRefactorability` and `@CodeRefactorability`, for which the user can turn on or off each one of the assertions $\phi_I, \ldots, \phi_{IV}$ and $\phi_{VI}, \ldots, \phi_{XII}$. These annotations can be configured to reflect properties whose combination deemed important to supporting additional refactoring actions, and of course for experimentation. For example, for *Pull Up Method* [11] verifying just two out of the four conditions defined for `@MovableMethod` (namely, ϕ_I and ϕ_{III}) should suffice. However, mapping the entire catalog of refactorings is a topic left for future work.

5.3 Implementation Notes

To provide language support that can prevent refactorability decay requires checking that code changes do not invalidate the conditions indicated by the annotations. When the conditions no longer hold true, the developer can be alerted and asked to adjust either the code (e.g., revert changes) or the invalid annotations (e.g., remove them).

One implementation strategy would be to integrate the annotation processor with an IDE, such as IntelliJ. When the annotation processor is enabled in the IDE, the conditions can be checked every time the code is compiled. This strategy has the advantage of potentially reusing the IDE's existing refactoring engine for some of the checks. The disadvantage is that it is difficult to keep the desired semantics aligned with what IntelliJ actually implements. For example, IntelliJ diverges from our approach in permitting EM when there exists a path without a return, and in ignoring object locks.[1]

Another implementation strategy would be to integrate the annotation processor with a version control system, such as GitHub. For example, the annotation processor can be invoked via *GitHub Actions*. This has the advantage of checking the conditions every time the code is checked-in, requiring the developer to adjust the code or the annotations before merging. In this case, the processor may be able to compare the code fragment against previous versions of the code and if the code was changed issue a "semantics-may-have-changed" warning even when the assertions hold (discussed in more detail in

[1] The actual behavior is IDE-specific. For example, in contrast to IntelliJ, the refactoring tool of Eclipse does alert the developer when there exists a path without a return.

Sect. 8.2).

Yet another implementation strategy, and the one we actually use in the prototyped implementation of the Corean library, is to use Java's annotation processing mechanism to implement our own precondition checking.[2] This enables us to scan and process the code during compilation independently of which IDE or version control is used. The advantage is rapid prototyping and ease of experimentation with different IDEs. The disadvantage is the duplicated checks with respect to existing refactoring engines.

[2] https://github.com/refactorability

Chapter 6

Methodology

In Sect. 3.4, we saw how the use of the new annotations can warn about, and prevent, refactorability decay. In this chapter, we will examine more thoroughly the various situations that arise in using the annotations, and the possibilities that exist in each situation.

6.1 Movable Code

This example shows the preparatory stage, preventing the formation of decay, and performing the refactoring.

Let's look again at List. 3.4 (without the yellow lines), describing the following usage scenario:

Developer δ_1 notices that the method contains a fragment of code that runs a CMD command with parameters, and saves the output. He understands that there is a chance that other developers will also need this capability elsewhere in the further development of the project. In this situation, developer δ_1 has 3 options.

1. To perform the full refactoring. This option requires:
   - Choosing a name for the new method.
   - Deciding on the parameters to be passed.
   - Performing the extraction.
   - Finding a suitable class for the new method

2. To document the useful code. This option has the following disadvantages:

    - There is no guarantee that in practice, the useful fragment can be extracted, without an understanding of the surrounding code.

    - There is no guarantee that during the life of the project, there will be no changes to the useful code fragment that will make extracting it in the future difficult.

3. To mark the useful code fragment with the `@MovableCode` annotation.

If at time τ_1, developer δ_1 is not interested in investing more time and effort in performing a full refactoring, we suggest that he use the `@MovableCode` annotation. In this case, after the annotation marking (as shown in List. 6.1), a compilation error, as shown in List. 6.2, is received.

Developer δ_1 is very familiar with the code he is currently writing. Therefore, he understands that it is possible to make a small change in the style of the code fragment, and thus solve the problem described in the compilation error. The possibility of such a solution is illustrated in List. C.4 on page 84. After this change, the code is compiled.

At this point, the new annotation helped developer δ_1 to prepare the code for future refactoring (see Definition 4.2.2).

At the next stage, suppose that later in the life of the project, developer δ_2 wants to print the total number of errors and warnings that were generated in the command run. In order to achieve this, he added the lines marked in yellow in figure List. 6.3. This change causes a compilation error, that is displayed in List. 6.4.

In this situation, developer δ_2 can choose one of the following three options:

1. If possible, he can fix the new code so that the problem causing the compilation error is solved.

2. He can delete the new code.

3. He can delete the annotation, and thus declare that the code fragment is no longer intended for refactoring.

Listing 6.1: The useful code annotated with @MovableCode

```java
public class Source {
    ⋮ ~~~The beginning of the code omitted~~~
    private String results;
    ⋮ ~~~Part of the code omitted~~~
    @MovableCode(Description = "Run a command and save its output")
    private void foo(String[] commandsArray, String[] paramsArray) {
        if (commandsArray.length == paramsArray.length) {
          for (int i=0;i<commandsArray.length;i++) {
                if ((commandsArray[i].matches(".*[{`%].*")) ||
                    ↪ (paramsArray[i].matches(".*[{`%].*"))) continue;
                String p = paramsArray[i];
                String[] words = p.split("\\s+");
                if (words.length > 2) continue;
                String[] commands = {commandsArray[i], paramsArray[i]};
                /*@MovableBegin*/
                Runtime rt = Runtime.getRuntime();
                Process proc = rt.exec(commands);
                BufferedReader stdInput = new BufferedReader(new
                    ↪ InputStreamReader(proc.getInputStream()));
                String outputLine = stdInput.readLine();
                while (outputLine != null) {
                  results += outputLine;
                  outputLine = stdInput.readLine();
                }
                /*@MovableEnd*/
          }
        }
    }
    ⋮ ~~~The rest of the code omitted~~~
}
```

Listing 6.2: Error message caused by @MovableCode annotation

```
java: Code fragment cannot be extracted and then moved. A code fragment marked
    ↪ as Movable in method foo of the class Source, contains assignment to
    ↪ instance variable (results).
```

Listing 6.3: List. C.4 after changes by developer δ_2

```
1  public class Source {
2      ⋮ ∼∼∼The beginning of the code omitted∼∼∼
3      private String results;
4      ⋮ ∼∼∼Part of the code omitted∼∼∼
5      @MovableCode(Description = "Run a command and save its output")
6      private void foo(String[] commandsArray, String[] paramsArray) {
7          if (commandsArray.length == paramsArray.length) {
8              int numOfErrors = 0; int numOfWarnings = 0;
9              for (int i=0;i<commandsArray.length;i++) {
10                 if ((commandsArray[i].matches(".*[{`%].*")) ||
                       ↪ (paramsArray[i].matches(".*[{`%].*"))) continue;
11                 String p = paramsArray[i];
12                 String[] words = p.split("\\s+");
13                 if (words.length > 2) continue;
14                 String[] commands = {commandsArray[i], paramsArray[i]};
15                 /*@MovableBegin*/
16                 Runtime rt = Runtime.getRuntime();
17                 Process proc = rt.exec(commands);
18                 BufferedReader stdInput = new BufferedReader(new
                       ↪ InputStreamReader(proc.getInputStream()));
19                 String outputLine = stdInput.readLine();
20                 String lineResult = "";
21                 while (outputLine != null) {
22                     if (outputLine.contains("error")) numOfErrors++;
23                     if (outputLine.contains("warning")) numOfWarnings++;
24                     lineResult += outputLine;
25                     outputLine = stdInput.readLine();
26                 }
27                 /*@MovableEnd*/
28                 results += lineResult;
29                 System.out.println("There were " + numOfErrors + " errors and " +
                       ↪ numOfWarnings + " warnings");
30             }
31         }
32     }
33     ⋮ ∼∼∼The rest of the code omitted∼∼∼
34 }
```

Listing 6.4: Error message caused by @MovableCode annotation

```
1  java: Code fragment cannot be extracted. A code fragment marked as Movable in
       ↪ method foo of the class Source, contains contains 2 variables
       ↪ (numOfErrors and numOfWarnings) which change their value in the
       ↪ extractable fragment but are used outside this fragment. Multiple
       ↪ variables cannot be the return value of an extracted method.
```

Option 3 is suitable for cases in which the change in the code is necessary, even at the cost of making the code non-reusable.

Suppose developer δ_2 chose option 1 and moves the new code out of the annotated section, as shown in List. C.5 on page 85.

At this stage, the annotation helped maintain the annotated code with the characteristics of "refactoring-ready" (Definition 4.1.1) throughout the life of the project.

In the last stage, if in practice the need arises for this capability in another place in the project, developer δ_3 can use the annotations to find that such a capability has already been realized in the project (as explained below in Sect. 8.1), and can also use the annotation to identify the relevant passage within the lengthy method.

Now it remains to actually extract the useful fragment and to move the new method to a suitable class, that will be accessible from both the original method and the additional place in the project. The annotation assures developer δ_3 that the fact that he is not deeply familiar with all the code of the original method and class, will not make it difficult for him to perform the necessary refactoring operations.

6.2 Movable Method

This example shows the preparatory stage, preventing the formation of decay, and performing the refactoring.

Developer δ_1 created some class `C`, in the framework of which there arises a need to check whether the file `fileA` contains a particular string. To accomplish this, developer δ_1 added the `isContain` **private** method, as shown in List. 6.5.

Developer δ_1 notices that this method performs a general operation that may be useful in other places in the project in the future.

In this situation, there are the following three options:

1. To make the `isContain` method **public**. This option is not recommended, as it would compromise the *Single Responsibility* of class `C`.

2. To move the `isContain` method to an appropriate class. This option requires searching for such a class, or possibly creating a new class.

Listing 6.5: A useful **private** method

```
1  public class C {
2      ⋮ ~~~The beginning of the code omitted~~~
3      boolean b;
4      ⋮ ~~~Part of the code omitted~~~
5      void foo() {
6          ⋮ ~~~The beginning of the code omitted~~~
7          isContain("fileA", "some string");
8          ⋮ ~~~The rest of the code omitted~~~
9      }
10
11     private void isContain(String filePath, String text){
12         b = false;
13         Scanner scanner = new Scanner(new File(filePath));
14         while (scanner.hasNextLine()) {
15             String line = scanner.nextLine();
16             if (line.contains(text)) {
17                 b = true;
18                 break;
19             }
20         }
21     }
22     ⋮ ~~~The rest of the code omitted~~~
23 }
```

Listing 6.6: Error message after annotating List. 6.5 with @MovableMethod

```
1 java: The method isContains of the class C, contains assignment to instance
    ↪ variable. This can make difficult to move this method to another class.
```

Listing 6.7: Error message caused by the changes made by developer δ_2

```
1 java: The method isContains of the class C, is locked on "this". This can cause
    ↪ a change in behavior if the method is moved to another class.
```

3. To mark the method with the @MovableMethod annotation.

Suppose that developer δ_1 chose to mark with annotation (option 3), as a result of which we receive the compilation error message displayed in List. 6.6.

It is likely that developer δ_1, who created the class C and the isContain method, will have no difficulty in changing the style of the code (for example, as shown in List. C.6 on page 86) and thus fixing the problem that caused the compilation error.

In this case, as well, the annotation @MovableMethod annotation helped developer δ_1 to prepare the isContain method for future refactoring (see Definition 4.2.2).

In the next stage, suppose that developer δ_2 wants to add a new bar method to class C, that also accesses the file fileA. To avoid concurrent access to the file, developer δ_2 makes the isContain and bar methods synchronized (as shown in List. C.7 on page 87).

In this situation, both methods are locked on this object of class C. Therefore, if we transfer the isContain method to another class, its this will change, and as a result, there will no longer be a mutual locking between the isContain and bar methods.

The change that developer developer δ_2 makes in isContain annotated method causes a compilation error, that is displayed in List. 6.7.

As in the previous example, developer δ_2 has 3 options to handle the error. Assume that developer δ_2 has chosen to change the code so that the isContain method does not lock on the current object (for example, as shown in List. C.8 on page 88).

At this stage, the annotation prevented the formation of refactorability decay which would have made it difficult for the safe execution of MM in the future.

In the last stage, if the need arises for such a capability in another place in the project, the annotation can help to find the **private** method that performs the required action (as

Listing 6.8: List. 3.9 with the @MovableCode annotation added

```
1  @MovableCode(Description = "Calculate␣the␣balance␣between␣{␣and␣}")
2  public static String apply(String input) {
3          ⋮ ~~~The beginning of the code omitted~~~
4          String c = matcher.replaceAll("");
5          /*@MovableBegin*/
6          long diff = c.chars().filter(ch->ch=='{').count() -
              ↪ c.chars().filter(ch->ch=='}').count();
7          /*@MovableEnd*/
8          ⋮ ~~~The rest of the code omitted~~~
9  }
```

explained below in Sect. 8.1). In particular, the annotation ensures that even without a deep understanding of all the code of class C, the isContain method can be easily moved to another class, and this move will not change the original behavior of the code.

6.3 Annotation of Code that is Already Ready for Refraction

Let's look at List. 3.9 again; let's say that developer δ_1 notices that the apply method contains a useful fragment of code that calculates the balance between the brackets. Instead of refactoring, he can mark the relevant lines with the @MovableCode annotation, as shown in List. 6.8.

In this case, the code is compilable even after the annotation marking, since the relevant section is already ready for refactoring in the original form. In this case, the operation required from developer δ_1 is similar to the documentation of the code.

In the future, if in practice the need for this capability arises in another place in the project, the annotation will help the developer δ_3 to find and identify the relevant code, and give him confidence that EM + MM refactoring can be performed on this fragment, even without a deep understanding all the surrounding code.

Listing 6.9: List. 3.5 with the @ExtractableCode annotation added

```
1  @ExtractableCode(Description = "Calculate␣the␣balance␣between␣{␣and␣}")
2  int convertSpecialChar(StringBuilder sb, char[] c, int start, FORMAT_M format) {
3    ⋮ ∼∼∼The beginning of the code omitted∼∼∼
4    /*@ExtractableBegin*/
5    while ((i<c.length) && (braceLevel>0) && (c[i]!='\\')) {
6      if (c[i] == '}') {
7        braceLevel--;
8      } else if (c[i] == '{') {
9        braceLevel++;
10     }
11     /*@ExtractableEnd*/
12     i = convertNonControl(c, i, sb, format);
13   }
14   ⋮ ∼∼∼The rest of the code omitted∼∼∼
15 }
```

6.4 Code that is Difficult to Refactor

Let's look at List. 3.5 again. Suppose that developer δ_1 detects that the code contains a useful fragment, and marks it with the @ExtractableCode annotation, as shown in List. 6.9.

As we saw in Sect. 3.4, the annotation results in a compilation error, as shown in List. 3.14. In this situation, for compilation, developer δ_1 has two options:

1. To mark the entire block of the **while** loop, and move the line: *"i = convertNonControl(c, i, sb, format)"* out of the loop since this operation is not related to the calculation of the balance between brackets.

2. To delete the annotation.

Is it possible to easily take the line *"i = convertNonControl(c, i, sb, format)"* out of the **while** loop? Developer δ_1, who wrote the code, is in the best position to answer this question!

Suppose developer δ_1 decided that the required modification was complex, and as a result, he decided to delete the annotation. The lack of marking in the annotation can give an indication to developer δ_3 that an attempt to perform EM on this fragment, may be complex and not worthwhile.

Therefore, if the project development team makes sure (for example, at the code review level) to mark all the relevant fragments with new annotations, then the lack of markup can help developer δ_3 conclude that it is not worth trying to refactor a fragment that was not marked by the original developer of the code.

Without the use of the new annotations, the current situation in many projects is that at first sight, it is not always clear when it is easy, and when it is complex to perform, refactoring. As a result of this lack of clarity, developer δ_3 is liable to may fail to perform simple refactoring, and, on the other hand, may spend time undue time trying to perform the complex refactoring.

Chapter 7

Evaluation

We conducted a user study to assess the potential usefulness of the new annotations. The refactoring annotations have two types of “clients”: developer δ_1, who must understand what the annotations mean and how they trigger compilation errors; and developer δ_3, who needs only understand how to interpret them in refactoring an annotated code fragment.

The study included synthetic refactoring tasks grouped into pairs of comparable difficulty [23]: $\langle A_1; B_1\rangle, \langle A_2; B_2\rangle, \ldots$ Each pair of tasks contained two methods with a useful fragment of code that should be extracted using the EM refactoring action and should then be moved to another class using the MM refactoring action. Each method was 10–60 lines of code in size, and the useful fragment 2–25 lines of code. In all the tasks some level of manual change to the code was required in order to complete the refactoring action.

We showed each pair of tasks $\langle A_i; B_i\rangle$ to two experienced developers, who confirmed that the necessary changes in A_i and B_i seemed comparable in terms of the level of difficulty and complexity[1].

7.1 Simulating Developer δ_1

Four developers were asked to familiarize themselves with the code of the tasks up to a level, they felt comfortable making changes as if they had written the code themselves. The developers received a short presentation explaining how to use the new refactoring

[1] For example, see Table D.1 on page 94

annotations. They also participated in a hands-on tutorial allowing them to experiment with the new annotations on several code examples.

Experiment: Each developer was then asked to perform one of two things on each of the tasks:

- Either annotate with `@MovableCode` a useful code fragment but not refactor it. We denote this action by α for *annotate*;
- Or fully refactor a useful code fragment without annotating it. We denote this action by ρ for *refactor*.

In each pair of tasks, two tasks with code without annotations were given as tasks A_i and B_i.[2] Some of the developers were asked to perform $\alpha(A_i)$, i.e., to locate and annotate a useful code fragment but not refactor it, and also to perform $\rho(B_i)$, i.e., refactor a useful code fragment without the use of annotations. Other developers were asked to perform $\rho(A_i)$ and $\alpha(B_i)$.

The developers were asked to perform either $\langle\alpha(A_i);\rho(B_i)\rangle$, $\langle\alpha(B_i);\rho(A_i)\rangle$, $\langle\rho(B_i);\alpha(A_i)\rangle$, or $\langle\rho(A_i);\alpha(B_i)\rangle$. The experiment was conducted in this way in order to reduce the learning effect between the tasks, and to cancel out any possible differences between the pair of tasks $\langle A_i; B_i\rangle$.

After completing the tasks, the participants were asked four open questions:

[Q1] How difficult was it to annotate the code (including getting it to compile again)?

[Q2] How difficult was it to refactor the unannotated code?

[Q3] What are the advantages and disadvantages of using annotations compared to full refactoring?

[Q4] Would you use such annotations if they were available in your programming language?

[2] List. D.1 and List. D.3 (on pages 90 and 92) are examples of A_i and B_i, respectively.

7.2 Simulating Developer δ_3

Eight developers that were not previously exposed to the code received just the short presentation explaining how to interpret the new refactoring annotations, without the tutorial part.

Experiment: Each developer was then given a task either in its original form without annotations, or in its annotated form in which the useful code fragment was annotated (denoted by "*")[3]. The developer was asked to extract to a method a fragment of code that does the desired task and then move the new method to a specific class. To reduce the impact of the learning effect between tasks on the results, the developers performed the refactoring tasks in random order; i.e., either $\langle \rho(A_i^*); \rho(B_i) \rangle$, $\langle \rho(B_i); \rho(A_i^*) \rangle$, $\langle \rho(B_i^*); \rho(A_i) \rangle$, or $\langle \rho(A_i); \rho(B_i^*) \rangle$.

After completing the tasks, the participants were asked four open questions:

[Q5] Was the meaning of the annotation clear?

[Q6] What were the difficulties in refactoring annotated code versus refactoring unannotated code?

[Q7] What is your level of confidence in the correctness of the refactoring of annotated code versus the refactoring of unannotated code?

[Q8] Would you have liked such annotations to be used in the projects you are maintaining?

7.3 Results

With regards to changes they made in the code, the developers simulating developer δ_1 described them as simple to perform. In contrast, the developers simulating developer δ_3 described these changes as difficult to make. Some of the participants noted that in a "real situation" they would give up on the idea of reusing (unannotated) code, because of

[3] List. D.2, on page 91, shows an annotated version of List. D.1, as an example of $A_i^* = \alpha(A_i)$.

the difficulty involved, and because of their uncertainty that the refactoring task would be performed correctly.

Specifically, in the first experiment that was designed to simulate developer δ_1:

[Q1] All of the participants testified that it was easy to annotate the code. Some of the participants said that the effort was similar to documentation.

[Q2] All of the participants reported that refactoring unannotated code demanded significantly more effort. Some cited the need to choose a meaningful name for the method and to find an appropriate class for it as a difficulty. One participant noted that compilation errors triggered by the annotations pointed more clearly to the code changes that were required, compared to standard compilation errors.

[Q3] All of the participants agreed that the use of annotations was easier and took less time than full refactoring, but noted that the size of the annotated code seemed longer than refactored code. Some noted that the use of annotations eliminated the need for creating unnecessary methods and classes.

[Q4] All of the participants said that in case they do not want to perform *preventive refactoring* in advance, they would likely use such annotations if available.

In the second experiment that was designed to simulate developer δ_3:

[Q5] All of the participants testified that the meaning of the annotation was clear. Some of the participants also noted that it was clearer where the relevant code begins and ends, and what refactoring action is recommended.

[Q6] All of the participants reported that the annotated code was clearer, enabling them to focus on the relevant parts, and requiring fewer changes to the code compared to unannotated code. Some cited the dependency of the relevant parts of the code on other parts of the method as the source of difficulty in refactoring unannotated code.

[Q7] The level of confidence in the correctness of the refactoring actions was given on a scale from 1 (low confidence) to 10 (high confidence). On average, the confidence level was 8 for annotated code, and 4.5 for unannotated code.

[Q8] All the participants said that they would welcome such annotations in their own projects, because these annotations would help them focus on the relevant parts of the code when refactoring.

It is interesting to note that all the developers simulating developer δ_1 completed the task without errors. In contrast, in the tasks done on the unannotated code, errors were made by developers simulating developer δ_3 in 25% of the tasks on average[4].

A possible explanation might be the different preparation. The developers simulating developer δ_1 knew the code of the whole method, because they were given enough time to study the code, and therefore knew with confidence what needed to be changed, and whether the change affected the rest of the code of the method. In contrast, the developers simulating developer δ_3 were unfamiliar with the code, and this created uncertainty about the required changes and the effect of the changes on the rest of the code.

7.4 Threats to Validity

The experiment was done on a small group of developers, imperfectly mimicking actual refactoring practices, and hence the results should be interpreted with caution. However, all the 14 developers selected to participate in the trial were software engineers with 1–10 years of experience. Most of them with a degree in Computer Science, and therefore they do represent engineers in the software industry.

The code tasks were synthesized for the experiment. However, to increase the reliability of the experiment, we selected code fragments similar to the code we have encountered in real industry software. In addition, the two developers that reviewed the code fragments confirmed that similar code fragments can be found in real software.

The developers simulating developer δ_1 did not write the code themselves, but only got a chance to study the code. However, we assume that if these developers had written the code themselves, their level of understanding of the code and confidence in the changes made would have just been even higher.

[4] App. D.2, on page 95, shows a breakdown of the errors.

Chapter 8

Discussion

In this chapter, we discuss improvements and additional uses of annotations presented in Sect. 5.2. Also, we discuss possible follow-up research directions for the ideas of properties of a code fragment and shared responsibility for refactoring actions, presented in this study.

8.1 Another Possible Benefit of Using Annotations

To reuse code that appears in our project, we must find it. Studies [24] indicated that programmers frequently search for code, conducting an average of five search sessions, with 12 total queries, each workday. A large number of searches are done to find reusable code. In practice, in many cases, if the developer wants to know if a certain capability was implemented in the project, he uses the following search techniques:

- He searches for a class that is responsible for the area in question and examines if an appropriate method exists in this class;
- He searches the entire project by keywords.

The first technique will not help to find code fragments and **private** methods that are located in classes that are responsible for other areas.

Searching by keywords will not find code fragments that do not contain the keywords that the developer thought of. Alternatively, it is possible that the search will find a large number of results, and a great deal of time will be needed to examine all of them and

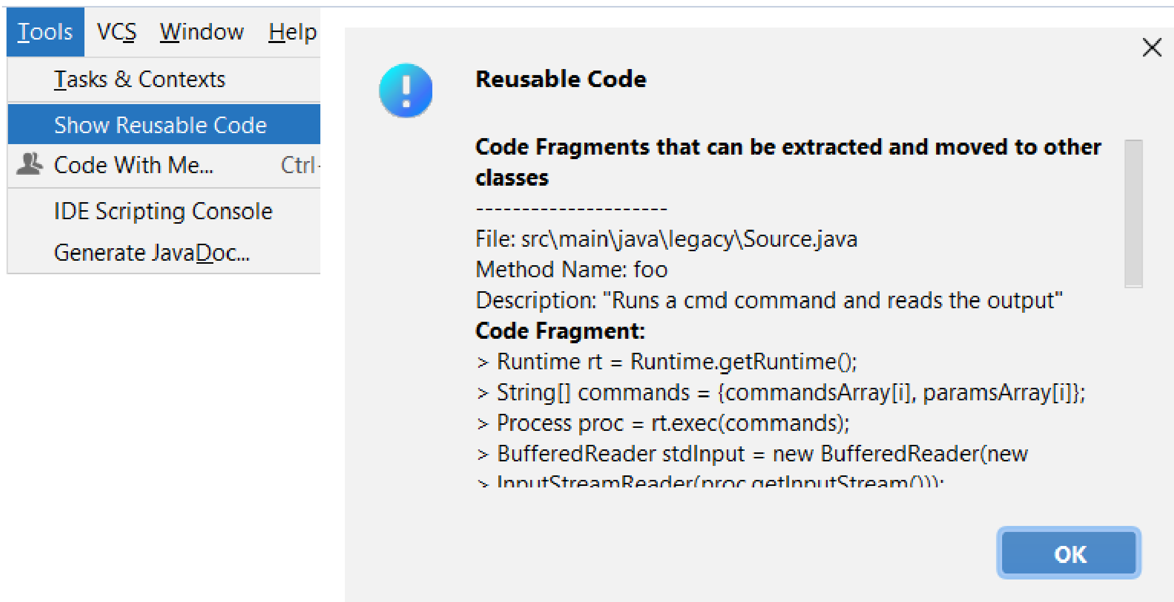

Figure 8.1: The option Show Reusable Code added to Tools by the plugin

filter out those irrelevant to the subject of reuse. The annotations that were presented in Sect. 5.2 make it possible to filter **private** methods and code that may be appropriate for reuse. In addition, every annotation contains a field named `Description`, that enables developer δ_1 to describe the action that the code fragment or the method performs.

We implemented an IntelliJ plugin that takes advantage of these annotations in order to display the code fragments and methods that may serve for reuse in a convenient and centralized way, despite the fact that most probably they would not have been found by means of a regular search. This plugin adds the option of Show Reusable Code to Tools (Fig. 8.1). Selecting this option displays a description of, and information about, the code fragments and methods marked by annotations, and therefore may serve for reuse. The results are divided into three parts:

- Methods that may be moved to another class.
- Code fragments that may be extracted into a separate method.
- Code fragments that may be extracted into a separate method and moved to another class.

For each annotated code, its location and description are displayed (Fig. 8.1).

8.2 Directions for Further Research

In this section, we will review the issues that have arisen, but a thorough investigation of them is beyond the scope of this study. We have left these issues for further research in the future.

8.2.1 Mapping the Entire Refactoring Catalog

In Sect. 1.3.2, we argued that refactoring actions can be classified according to the following criteria:

- The refactoring action consists mainly of σ-steps.
- The refactoring action consists mainly of μ-steps.
- The refactoring action comprises multiple σ-steps and multiple μ-steps,

All refactoring actions from the catalog can be broken down into steps, and for each step, it can be determined whether it is μ-steps or σ-steps. Accordingly, a recommendation can be formulated as to whether this should be performed by the developer δ_1, by the developer δ_3, or under joint responsibility. For example, a preliminary analysis for a Pull-Up Method reveals that the operation consists of the steps listed in Alg. 3.

Algorithm 3 *Pull Up Method*

[PU-1_μ] Finding the most suitable level for method M in the inheritance chain.

[PU-2_σ] Separate method M from the context of the source class C_1.

[PU-3_μ] Adjust calls to M.

[PU-4_σ] Verify that M's original behavior did not change.

[PU-5_μ] Pull method M up in the inheritance chain to the target class C_2.

The properties of the code that can make it difficult to perform steps [PU-2_σ] and [PU-4_σ] are:

- Assignment of a value to an instance variable not defined in the root class.
- Calling a method (of any type of access modifiers) defined in subclasses.

This is because it is possible that the instance variable and the method will not be accessible from the parent class to which we would like to pull the method.

8.2.2 Annotation for Self-contained Code that is Suitable for Reuse

EM and MM are useful refactoring actions that enable reuse of code that is already in the project. On the other hand, when we want to reuse the code that appears in another module, or code from Stack Overflow, a copy-paste operation is performed, followed by necessary adjustments [15, 25, 26]. If the code is self-contained, this is usually a simple operation. On the other hand, if the code intended to be copied depends on other elements in the source code, this can be complicated, or perhaps even completely not worthwhile. It is possible to define and test the usefulness of an annotation that will indicate that the code fragment (or method) intended for reuse is self-contained.

8.2.3 Possible Improvements to the Annotations Presented in Sect. 5.2

- The `@ExtractableCode` and `@MovableCode` annotations make it possible to mark only sequential code fragments. It is possible to study under what conditions discontinuous segments can be marked, while still maintaining the desired properties.
- The `@MovableMethod` annotation is used to mark useful methods that we might want to move to another class, but it is also used to mark auxiliary methods, that must be moved along with the useful method. It is possible to design annotation with greater granularity, which will allow marking auxiliary methods differently.
- In this study, when we defined the properties of code that can make refactoring actions difficult, we relied mainly on possible difficulties described in the scientific literature. It is possible to conduct extensive in-the-field research to examine to what extent, in practice, different code properties make it difficult for the developer δ_3 to perform different types of *corrective refactoring*. Depending on the results received, the list of desirable code properties for the desired types of refactoring actions can be updated.

8.2.4 Support for Refactorability Decay Prevention in Additional Languages

Our goal is for refactorability decay prevention support to be an integral part of programming languages. But in the meantime, to show the feasibility and test usefulness and effectiveness, we used the *Annotation Processor* for JAVA. It is possible to examine what mechanisms can integrate this capability into other languages as well. For example, it seems that the *#pragma* mechanism can make it possible to add this capability to C++.

8.2.5 Semantic Changes in Reusable Code

The annotations that we have presented protect reusable code from changes that will make it difficult to perform refactoring in the future. But it is still possible that during the lifetime of the project, developer δ_2 will make changes to the code that will not cause difficulty in refactoring, but will change the semantics of the code. In these cases, it is important to check to ensure that the code still performs the action declared by the annotation.

It is possible to integrate the annotations with version control, and to detect that changes have been made to the annotated code relative to the previous version (for example, the test will be performed in *GitHub Actions* or *Git Hooks*). In order to treat cases of semantic changes in annotated code, two approaches can be taken:

1. Allow developer δ_2 to make the change, but to display a warning next to the modified annotated code. This warning will inform developer δ_3 that in order to reuse the annotated code, he must proceed with extreme caution, and ensure that the code performs the declared operation.

2. Allow developer δ_2 to perform a commit that makes a semantic change in the annotated code only if developer δ_2 confirms that the change does not affect the declared operation of the annotated code, and that even after the change, the code remains reusable. For example, this confirmation can be done using a special string that will be included in the commit message.

Chapter 9

Conclusion

We present a new approach to refactoring in which code annotations capture and maintain preconditions necessary for carrying out a refactoring action. These annotations allow the code's original developer δ_1 to document refactoring-relevant information so that, at some point in the future when the code starts to smell, another developer δ_3, guided by the annotations, can performs the refactoring action itself with ease and confidence.

The annotations provide developer δ_1 with a channel for communicating refactoring information to developer δ_3, as well as a means for reducing the risk of refactorability decay between time τ_1 and time τ_3. Unlike documentation that does not promise ease of reuse and often becomes out of date as soon as changes are made to the code, the annotated code lends itself to reuse and resists changes at time τ_2 that can break refactoring, thus increasing the chances for it being reused at time τ_3.

The use of annotations imposes on developer δ_1 less effort than full *preventive refactoring* at time τ_1. It also reduces developer δ_3's efforts at time τ_3 compared to *corrective refactoring* on code that is not annotated. Meanwhile, the use of these annotations increases the willingness of developers to perform refactoring on unfamiliar code, and improves their confidence in the correctness of the refactoring action.

Our approach to refactoring offers a new separation of refactoring concerns that splits the refactoring action into two stages and the responsibility for carrying it out into two roles. By doing so, the approach supports a refactoring process that distributes the costs of refactoring over time, thus encouraging more refactorings than the standard process. We hope that disciplined use of our approach would be a step toward a new refactoring-aware programming style.

Appendix A

Developer Guide

The guide describes the internal structure of the Corean library. It can be used by developers who wish to continue to develop the library, for example, to add to the annotations additional properties to be tested or to add new annotations.

A.1 Overview of the Class Structure

In this section, we will briefly review the structure of the core packages and classes of the Corean library. The class diagram of the library appears in Fig. A.1. The Java Doc of the library appears at `https://refactorability.github.io/Corean-Java-Doc/`

A.1.1 The `annotations` Package

The `annotations` package contains a definition of five annotations. Each annotation declares certain properties that exist in the annotated method or code fragment.

A.1.2 The `processors` Package

The `processors` package contains a definition of classes that handle compilation-time testing for the methods marked with annotations.

`PropertyProcessor`

The `PropertyProcessor` class inherits from the `AbstractProcessor` class and

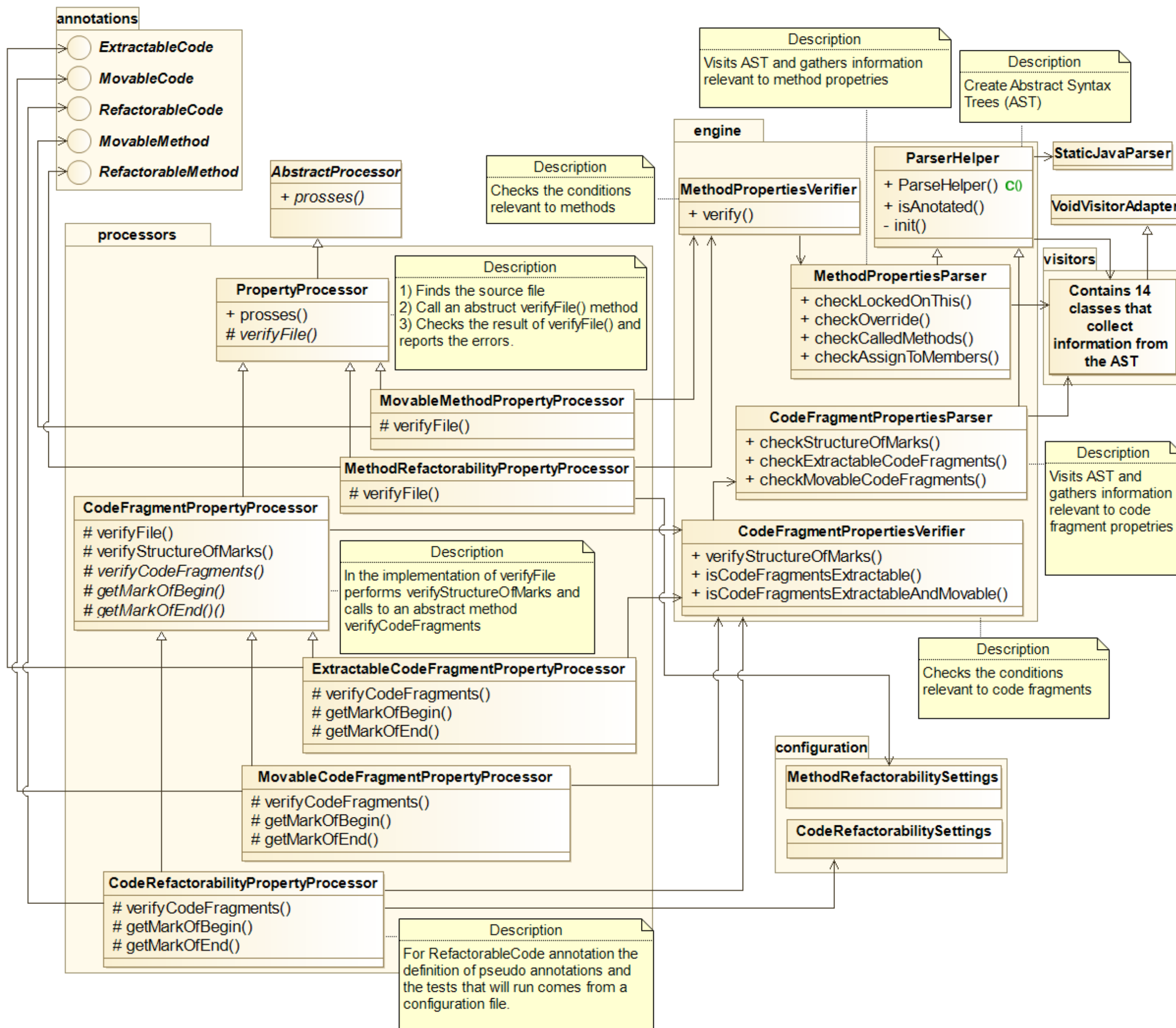

Figure A.1: The class diagram of Corean

serves as a root class for classes that verify that the code does not violate certain properties. This class overrides the `process` method and performs three operations:

1. Finds the source file that contains the appropriate annotation.
2. Checks that there is no violation of properties expected from the annotated code. This test is performed by calling the **abstract** method `verifyFile`.
3. Checks the result of `verifyFile` and reports compilation errors.

`MovableMethodPropertyProcessor`

The `MovableMethodPropertyProcessor` class inherits from the `PropertyProcessor` class, and checks that there is no property violation for methods that are annotated by `@MovableMethod`. This class implements the `verifyFile` method to check that all four properties described in Sect. 5.2.1 are present. The actual test is performed by calling the `verify` method of the class `MethodPropertiesVerifier` and passing a parameter indicating that all four properties must be checked.

`MethodRefactorabilityPropertyProcessor`

The `MethodRefactorabilityPropertyProcessor` class inherits from the `PropertyProcessor` class and checks that there is no property violation for methods annotated by `@RefactorableMethod`. This class implements the `verifyFile` method such that it checks that the properties defined in the *refactorability_configuration.json* file for method refactorability are present. The actual test is performed by calling the `verify` method of the class `MethodPropertiesVerifier`, and passing a parameter indicating which tests should be performed.

`CodeFragmentPropertyProcessor`

The `CodeFragmentPropertyProcessor` class inherits from the `PropertyProcessor` class, and serves as a root class for classes that verify that a code fragment does not violate certain properties. This class adds the ability to define a new marking for the beginning of a code fragment and for the end of a fragment of code, and checks that each method marked with an appropriate annotation contains a valid structure

of code fragment marks. The class defines **abstract** methods getMarkOfBegin and getMarkOfEnd, by way of which the marks for the beginning and end of a fragment are defined, and defines an **abstract** verifyCodeFragments method that checks that the marked code fragments do not violate the desired properties. In addition, the verifyStructureOfMarks method, which checks that the fragment mark structure is valid, is defined and implemented. The test is done by checking the following conditions:

1. There is at least one beginning-of-a-fragment mark.
2. An end-of-a-fragment mark comes after each beginning-of-a-fragment mark.
3. There are no two consecutive beginning-of-fragment marks or end-of-fragment marks.
4. Each end-of-a-fragment mark follows a beginning-of-a-fragment mark.

The verifyStructureOfMarks method of the CodeFragmentPropertiesVerifier class is used to perform these tests. The CodeFragmentPropertyProcessor class implements the verifyFile method as follows: First, the method verifyStructureOfMarks, which checks the validity of the marking structure, is called, and then the **abstract** method verifyCodeFragments that will be implemented in inheriting classes, and checks that the marked code fragments do not violate the desired conditions.

ExtractableCodeFragmentPropertyProcessor

The ExtractableCodeFragmentPropertyProcessor class inherits from the CodeFragmentPropertyProcessor class, and checks that there is no property violation for methods annotated by @ExtractableCode. This class implements the verifyCodeFragments method so that the four properties ϕ_{VI} - ϕ_{IX} described in Sect. 5.2.2 are met. The actual test is performed by calling the isCodeFragmentsExtractable method of the CodeFragmentPropertiesVerifier class and passing a parameter specifying that all four properties must be checked.

MovableCodeFragmentPropertyProcessor

The MovableCodeFragmentPropertyProcessor class inherits from the

`CodeFragmentPropertyProcessor` class, and checks that there is no property violation for methods annotated by `@MovableCode`. This class implements the `verifyCodeFragments` method such that all the properties described in Sect. 5.2.3 are present. The actual test is performed by calling the `isCodeFragmentsExtractableAndMovable` method of the `CodeFragmentPropertiesVerifier` class.

`CodeRefactorabilityPropertyProcessor`

The `CodeRefactorabilityPropertyProcessor` class inherits from the `PropertyProcessor` class, and checks that there is no property violation for methods annotated by `@RefactorableCode`. This class implements the `verifyCodeFragments` method, such that the properties defined in the *refactorability_configuration.json* file for code refactorability are present. The actual test is performed by calling the `isCodeFragmentsExtractable` method of the `CodeFragmentPropertiesVerifier` class and passing a parameter that specifies which tests should be performed.

A.1.3 The `engine` Package

The `engine` package is responsible for actually performing checks that the code does not violate certain properties.

`MethodPropertiesVerifier`

The class `MethodPropertiesVerifier` is responsible for testing properties related to an entire method. The class accepts in the constructor a parameter that specifies which of the properties described in section Sect. 5.2.1 should be checked. The `verify` method performs the selected properties test and returns the overall test result. When actually performing the property check, the class uses the `MethodPropertiesParser` class.

`CodeFragmentPropertiesVerifier`

The `CodeFragmentPropertiesVerifier` class is responsible for all checks related to code fragments. The class defines and implements three methods:

`verifyStructureOfMarks` - Checks that the methods marked with an appropriate annotation also contain a valid code fragment marking structure.

`isCodeFragmentsExtractable` – Checks that the marked code fragments do not violate the properties defined in Sect. 5.2.2 (using a configuration variable, it is possible to define that only some properties are checked).

`isCodeFragmentsExtractableAndMovable` – Checks that the marked code fragments do not violate the properties defined in Sect. 5.2.3.

In the actual execution of the property check, the class uses the `CodeFragmentPropertiesParser` class.

`ParserHelper`

The `ParserHelper` class serves as a root class for classes that check for the existence of properties in the code by analyzing an abstract syntax tree (AST). This class creates the AST using the `StaticJavaParser` class, and checks which methods are annotated with the relevant annotation. The test is done by analyzing the information collected from the AST by classes from the `visitors` package.

`MethodPropertiesParser`

The class `MethodPropertiesParser` inherits from the `ParserHelper` class and adds methods that check for non-violation of properties in Sect. 5.2.1. The test is done by analyzing the information collected from the AST by classes from the `visitors` package.

`CodeFragmentPropertiesParser`

The `CodeFragmentPropertiesParser` class inherits from the `ParserHelper` class and adds checks related to code fragments. Here, too, the test is done by analyzing the information collected from the AST by classes from the `visitors` package.

A.1.4 The `visitors` Package

The `visitors` package contains classes that collect relevant information from within the AST. Each class inherits from the `VoidVisitorAdapter` class and overrides the `visit` method for nodes of a particular type (e.g. `MethodDeclaration`, `AssignExpr`,

ReturnStmt etc.). The method passes over all the nodes of this type and collects certain information.

A.2 Guidelines for Adding New Annotations and Tests

In order to add to the Corean library support for a new annotation that will test other properties, the following steps must be performed:

- Add a new interface that defines the annotation to the annotations package.
- Add a new class to the processors package to handle this annotation.

If the new annotation declares properties of a method, the new class should inherit from PropertyProcessor and implement the verifyFile method in which the new tests on the methods marked with the new annotation will be performed. If the new annotation declares properties of a code fragment, the new class should inherit from CodeFragmentPropertyProcessor and implement the **abstract** methods responsible for defining the marks for the beginning and end of a fragment. And as well, it must implement the **abstract** method verifyCodeFragments in which the new tests on the code fragments marked with the new annotation will be performed.

The tests themselves can be implemented in any way desired, but of course, the existing infrastructure in the engine and visitors packages can be used.

Appendix B

User Guide

This Guide is intended for users who want to use the annotations provided by the COREAN library in their project.

B.1 Adding the Corean Library to IntelliJ

In order to use the COREAN library from INTELLIJ, the developer must add the library to the project, and enable *Annotation Processors.* To do this, perform the following steps.

Adding the library to the project:

- Download the *collaborative-refactoring-annotations-0.0.1.jar* [1].
- Right-click on the project to which you want to add -> Open module settings. From the window that opens, select *Libraries* and click on *+ (New Project Library)*, select *Java*, and add the *collaborative-refactoring-annotations-0.0.1.jar* file that was downloaded in the previous step.

Enabling Annotation Processors:

Choose: *File* -> *Settings* -> *Build, Execution, Deployment* -> *Compiler* -> *Annotation Processors.* Check the *Enabling Annotation Processing* checkbox, and verify that *Obtain processors from project classpath* is selected.

If using MAVEN dependency must be added to *pom.xml* (List. B.1):

[1] https://github.com/refactorability/Collaborative-Refactoring-Annotations/blob/main/collaborative-refactoring-annotations-0.0.1.jar

Listing B.1: The dependency of COREAN library

```
1 <dependency>
2       <groupId>ac.collaborative.refactoring.annotations</groupId>
3       <artifactId>collaborative-refactoring-annotations</artifactId>
4       <version>0.0.1</version>
5 </dependency>
```

Listing B.2: The *refactorability_configuration.json* configuration file

```
{"codeRefactorability":{
    "listOfStatementsTest":,
    "continueBreakTest":,
    "returnTest":,
    "localVariableTest":,
    "annotationMeaning":,
    "annotationActionVerb":,
    "markOfBegin":,
    "markOfEnd":},
"methodRefactorability":{
    "overrideTest":,
    "lockTest":,
    "instanceVariableTest":,
    "callNotMoveableMethodTest":}
}
```

B.2 Using Configurable Annotations

In order to use `@RefactorableMethod` and `@RefactorableCode` annotations, place the configuration file *refactorability_configuration.json*[2] in the same folder as the project's *src* folder. The file allows you to configure the following settings for the `@RefactorableMethod` and `@RefactorableCode` annotations (List. B.2).

Settings for `@RefactorableCode`

- `listOfStatementsTest` - Determines whether to check property ϕ_{VI} from Sect. 5.2.2. Valid input: true/false.
- `continueBreakTest` - Determines whether to check property ϕ_{VII} from Sect. 5.2.2.
- `returnTest` - Determines whether to check property ϕ_{VIII} from Sect. 5.2.2.

[2] https://github.com/refactorability/Collaborative-Refactoring-Annotations/blob/main/refactorability_configuration.json

- `localVariableTest` - Determines whether to check property ϕ_{IX} from Sect. 5.2.2.
- `annotationMeaning` - The meaning of the annotation. E.g. "Refactorable", "Extractble", "Movable". Intended for the formulation of compilation error messages.
- `annotationActionVerb` - E.g. "refactored", "extracted", "moved". Intended for the formulation of compilation error messages.
- `markOfBegin` - The mark for the beginning of a fragment. E.g. "/*@Refactorable-Begin*/"
- `markOfEnd` - The mark for the end of a fragment. E.g. "/*@RefactorableEnd*/"

Settings for `@RefactorableMethod`

- `overrideTest` - Determines whether to check property ϕ_{IV} from Sect. 5.2.1.
- `lockTest` - Determines whether to check property ϕ_{II} from Sect. 5.2.1.
- `instanceVariableTest` - Determines whether to check property ϕ_{I} from Sect. 5.2.1.
- `callNotMoveableMethodTest` - Determines whether to check property ϕ_{III} from Sect. 5.2.1.

B.3 Limitations

For efficiency, the prototyped Corean library checks only the file that contains the annotated code, without loading other files. There are two limitations to this implementation decision: (1) checking that the method does not override another method relies on the `@Override` annotation; (2) checking that there are no assignments to an instance variable does not detect assignments to instance variables defined in the parent class.

In addition, there is the following limitation: In the file structure of the project, the *src* folder should be located in a folder that is the parent folder (or ancestral folder) of the *out*/*target* folders.

B.4 The ReusableCodeViewer Plugin

B.4.1 Adding the ReusableCodeViewer Plugin to IntelliJ

In order to manually add the ReusableCodeViewer plugin to IntelliJ, place the ReusableCodeViewer-1.0-SNAPSHOT.jar[3] file in the *ReusableCodeViewer\lib* folder under the *plugins* folder of the IntelliJ.

B.4.2 Limitations

The *src* folder should be located under the main project folder.

[3] https://github.com/refactorability/Collaborative-Refactoring-Annotations/blob/dd93ac399ddc267797f21ce699ef0e27b5346868/ReusableCodeViewer-1.0-SNAPSHOT.jar

Appendix C

Supplemental Examples

For completeness, this appendix provides the intermediate steps that were omitted for clarity in the examples shown in Chapter 3 and Chapter 6.

Listings C.1 to C.3 are supplemental to the Chapter 3. Listings C.4 to C.8 are supplemental to the Chapter 6.

Listing C.1: List. 3.1 after possible corrections of developer δ_1

```
1  public class JabRefGUI {
2    ⋮ ~~~The beginning of the code omitted~~~
3    private boolean correctedWindowPos;
4    ⋮ ~~~Part of the code omitted~~~
5    private void openWindow(Stage mainStage){
6      mainFrame.init();
7      GuiPreferences guiPreferences = preferencesService.getGuiPreferences();
8      boolean corrected;
9      // Restore window location and/or maximized state
10     if (guiPreferences.isWindowMaximized()) {
11        mainStage.setMaximized(true);
12        corrected = false;
13     } else if ((Screen.getScreens().size()==1) && isWindowPositionOutOfBounds()){
14        // corrects the Window, if it is outside the mainscreen
15        mainStage.setX(0);
16        mainStage.setY(0);
17        mainStage.setWidth(1024);
18        mainStage.setHeight(768);
19        corrected = true;
20     } else {
21        mainStage.setX(guiPreferences.getPositionX());
22        mainStage.setY(guiPreferences.getPositionY());
23        mainStage.setWidth(guiPreferences.getSizeX());
24        mainStage.setHeight(guiPreferences.getSizeY());
25        corrected = false;
26     }
27     correctedWindowPos = corrected;
28     ⋮ ~~~The rest of the code omitted~~~
29   }
30 }
```

Listing C.2: The result of the extraction with ECLIPSE

```
1  private void extracted(String value) {
2     int braceCount = 0;
3     for (int index = 0; index < value.length(); index++) {
4          char charAtIndex = value.charAt(index);
5          if (charAtIndex == '{') {
6             braceCount++;
7          } else if (charAtIndex == '}') {
8             braceCount--;
9          }
10         if (braceCount < 0) {
11            return true;
12         }
13    }
14 }
```

Listing C.3: The result of the extraction with INTELLIJ

```
1  private boolean hasNegativeBraceCount(String value) {
2     if (extracted(value)) return true;
3     return false;
4  }
5
6  private static boolean extracted(String value) {
7     int braceCount = 0;
8     for (int index = 0; index < value.length(); index++) {
9          char charAtIndex = value.charAt(index);
10         if (charAtIndex == '{') {
11                braceCount++;
12         } else if (charAtIndex == '}') {
13                braceCount--;
14         }
15         if (braceCount < 0) {
16                return true;
17         }
18    }
19    return false;
20 }
```

Listing C.4: List. 6.1 after the preparation step for EM and MM refactoring

```
1  public class Source {
2      ⋮ ~~~The beginning of the code omitted~~~
3      private String results;
4      ⋮ ~~~Part of the code omitted~~~
5      @MovableCode(Description = "Run␣a␣command␣and␣save␣its␣output")
6      private void foo(String[] commandsArray, String[] paramsArray) {
7          if (commandsArray.length == paramsArray.length) {
8            for (int i=0;i<commandsArray.length;i++) {
9                  if ((commandsArray[i].matches(".*[{`%].*")) ||
                       ↪ (paramsArray[i].matches(".*[{`%].*"))) continue;
10                 String p = paramsArray[i];
11                 String[] words = p.split("\\s+");
12                 if (words.length > 2) continue;
13                 String[] commands = {commandsArray[i], paramsArray[i]};
14                 /*@MovableBegin*/
15                 Runtime rt = Runtime.getRuntime();
16                 Process proc = rt.exec(commands);
17                 BufferedReader stdInput = new BufferedReader(new
                       ↪ InputStreamReader(proc.getInputStream()));
18                 String outputLine = stdInput.readLine();
19                 String lineResult = "";
20                 while (outputLine != null) {
21                   lineResult += outputLine;
22                   outputLine = stdInput.readLine();
23                 }
24                 /*@MovableEnd*/
25                 results += lineResult;
26           }
27         }
28     }
29     ⋮ ~~~The rest of the code omitted~~~
30 }
```

Listing C.5: List. 6.3 after possible modifications by developer δ_2 that do not cause refactorability decay

```
1  public class Source {
2      ⋮ ~~~The beginning of the code omitted~~~
3      private String results;
4      ⋮ ~~~Part of the code omitted~~~
5      @MovableCode(Description = "Run a command and save its output")
6      private void foo(String[] commandsArray, String[] paramsArray) {
7          if (commandsArray.length == paramsArray.length) {
8            int numOfErrors = 0; int numOfWarnings = 0;
9            for (int i=0;i<commandsArray.length;i++) {
10               if ((commandsArray[i].matches(".*[{`%].*")) ||
                     ↪ (paramsArray[i].matches(".*[{`%].*"))) continue;
11               String p = paramsArray[i];
12               String[] words = p.split("\\s+");
13               if (words.length > 2) continue;
14               String[] commands = {commandsArray[i], paramsArray[i]};
15               /*@MovableBegin*/
16               Runtime rt = Runtime.getRuntime();
17               Process proc = rt.exec(commands);
18               BufferedReader stdInput = new BufferedReader(new
                     ↪ InputStreamReader(proc.getInputStream()));
19               String outputLine = stdInput.readLine();
20               String lineResult = "";
21               while (outputLine != null) {
22                       lineResult += outputLine;
23                       outputLine = stdInput.readLine();
24               }
25               /*@MovableEnd*/
26               results += lineResult;
27               numOfErrors = lineResult.split("error", -1).length-1;
28               numOfWarnings = lineResult.split("warning", -1).length-1;
29               System.out.println("There were " + numOfErrors + " errors and " +
                     ↪ numOfWarnings + " warnings");
30           }
31         }
32     }
33     ⋮ ~~~The rest of the code omitted~~~
34 }
```

Listing C.6: List. 6.5 after the preparation step for MM refactoring

```
1  public class C {
2      ⋮ ~~~The beginning of the code omitted~~~
3      boolean b;
4      ⋮ ~~~Part of the code omitted~~~
5      void foo() {
6          ⋮ ~~~The beginning of the code omitted~~~
7          b = isContain("fileA", "some␣string");
8          ⋮ ~~~The rest of the code omitted~~~
9      }
10
11     @MovableMethod(Description = "Checks␣whether␣the␣file␣contains␣the␣string")
12     private boolean isContain(String filePath, String text) {
13             Scanner scanner = new Scanner(new File(filePath));
14             while (scanner.hasNextLine()) {
15                     String line = scanner.nextLine();
16                     if (line.contains(text)) {
17                             return true;
18                     }
19             }
20             return false;
21     }
22     ⋮ ~~~The rest of the code omitted~~~
23 }
```

Listing C.7: List. C.6 after the changes made by developer δ_2

```
1  public class C {
2          ⋮ ∼∼∼The beginning of the code omitted∼∼∼
3          boolean b;
4          ⋮ ∼∼∼Part of the code omitted∼∼∼
5          void foo() {
6                  ⋮ ∼∼∼The beginning of the code omitted∼∼∼
7                  b = isContain("fileA", "some string");
8                  ⋮ ∼∼∼The rest of the code omitted∼∼∼
9          }
10
11         @MovableMethod(Description = "Checks whether the file contains the string")
12         private synchronized boolean isContain(String filePath, String text){
13            Scanner scanner = new Scanner(new File(filePath));
14            while (scanner.hasNextLine()) {
15                    String line = scanner.nextLine();
16                    if (line.contains(text)) {
17                            return true;
18                    }
19            }
20            return false;
21         }
22
23         synchronized void bar(){
24                 ⋮ ∼∼∼The beginning of the code omitted∼∼∼
25                 //Code that accesses file "fileA"
26                 ⋮ ∼∼∼The rest of the code omitted∼∼∼
27         }
28         ⋮ ∼∼∼The rest of the code omitted∼∼∼
29 }
```

Listing C.8: List. C.7 after possible modifications by developer δ_2 that do not cause refactorability decay

```
1  public class C {
2          ⋮ ~~~The beginning of the code omitted~~~
3          boolean b;
4          ⋮ ~~~Part of the code omitted~~~
5          void foo() {
6                  ⋮ ~~~The beginning of the code omitted~~~
7                  synchronized(this)
8                  {
9                          b = isContain("fileA", "some string");
10                 }
11                 ⋮ ~~~The rest of the code omitted~~~
12         }
13
14         @MovableMethod(Description = "Checks whether the file contains the string")
15         private boolean isContain(String filePath, String text) {
16            Scanner scanner = new Scanner(new File(filePath));
17            while (scanner.hasNextLine()) {
18                    String line = scanner.nextLine();
19                    if (line.contains(text)) {
20                            return true;
21                    }
22            }
23            return false;
24         }
25
26         synchronized void bar() {
27                 ⋮ ~~~The beginning of the code omitted~~~
28                 //Code that accesses file "fileA"
29                 ⋮ ~~~The rest of the code omitted~~~
30         }
31         ⋮ ~~~The rest of the code omitted~~~
32 }
```

Appendix D

Evaluation Tasks

This appendix provides details about a pair of tasks given in the experiment as part of the evaluation.

D.1 Examples of the Tasks

In this section, we'll present an example of a pair of tasks $\langle A_i; B_i \rangle$ in the regular version and $A_i^* = \alpha(A_i)$, $B_i^* = \alpha(B_i)$, in the version annotated with `@MovableCode`. For the pair of tasks in the regular version, we will compare the manual changes that must be made to the code of each of the tasks, to enable the execution of EM and MM.

The `foo` method in List. D.1 contains a useful fragment of code, which runs a CMD command and saves the output.

List. D.2 shows the same method after we annotated the useful fragment with `@MovableCode` annotation and made the changes in the code needed to compile it.

The `foo` method in List. D.3 contains a useful fragment of code, which reads the values of a column by a given name from the database.

List. D.4 shows the same method after we annotated the useful fragment with `@MovableCode` annotation and made the changes in the code needed to compile it.

Listing D.1: Refactoring Task A_i (without annotations)

```
1  package tasks;
2
3  import java.io.*;
4
5  public class Task_A {
6
7    public String getResults() {
8      return results;
9    }
10
11   public void setResults(String results) {
12     this.results = results;
13   }
14
15   String results = "";
16
17   public void foo(String[] commandsArray, String[] paramsArray) throws
       ↪ IOException {
18     if (commandsArray.length == paramsArray.length) {
19       for (int i = 0; i < commandsArray.length; i++) {
20         if ((commandsArray[i].matches(".*[{`%].*")) ||
             ↪ (paramsArray[i].matches(".*[{`%].*"))) continue;
21         String p = paramsArray[i];
22         String[] words = p.split("\\s+");
23         if (words.length > 2)
24           continue;
25         String[] commands = new String[2];
26         commands[0] = commandsArray[i];
27         commands[1] = paramsArray[i];
28         Runtime rt = Runtime.getRuntime();
29         Process proc = rt.exec(commands);
30         BufferedReader stdInput = new BufferedReader(new
             ↪ InputStreamReader(proc.getInputStream()));
31         if (stdInput != null) {
32           String s = stdInput.readLine();
33           while (s != null) {
34             results += s;
35             s = stdInput.readLine();
36           }
37         }
38       }
39     }
40   }
41 }
42
43
44
45
46 .
```

Listing D.2: Refactoring Task A_i^* (with annotations)

```java
package tasks;

import ac.collaborative.refactoring.annotations.MovableCode;
import java.io.*;

public class Task_A {
  public String getResults() {
    return results;
  }

  public void setResults(String results) {
    this.results = results;
  }

  String results = "";

  @MovableCode(Description = "Run a command and save its output")
  public void foo(String[] commandsArray, String[] paramsArray) throws
      IOException {
    if (commandsArray.length == paramsArray.length) {
      for (int i = 0; i < commandsArray.length; i++) {
        if ((commandsArray[i].matches(".*[{`%].*")) ||
            (paramsArray[i].matches(".*[{`%].*"))) continue;
        String p = paramsArray[i];
        String[] words = p.split("\\s+");
        if (words.length > 2)
          continue;
        String[] commands = new String[2];
        commands[0] = commandsArray[i];
        commands[1] = paramsArray[i];
        /*@MovableBegin*/
        Runtime rt = Runtime.getRuntime();
        Process proc = rt.exec(commands);
        BufferedReader stdInput = new BufferedReader(new
            InputStreamReader(proc.getInputStream()));
        String result = "";
        if(stdInput != null){
          String s = stdInput.readLine();
          while (s != null) {
            result += s;
            s = stdInput.readLine();
          }
        }
        /*@MovableEnd*/
        results+=result;
      }
    }
  }
}
```

Listing D.3: Refactoring Task B_i (without annotations)

```
package tasks;

import java.io.IOException;
import java.sql.*;

public class Task_B {

    public String getResults() {
        return results;
    }

    public void setResults(String results) {
        this.results = results;
    }

    String results = "";

    public void foo(String[] columnNames, String[] tablesNames, String
        ↪ mysqlUrl, String delimiter) throws IOException, SQLException {
        if (tablesNames.length == columnNames.length) {
            for (int i = 0; i < columnNames.length; i++) {
                if ((columnNames[i].contains("Demo")) ||
                    ↪ (tablesNames[i].contains("Demo"))) continue;
                String[] words = columnNames[i].split("\\s+");
                if (words.length > 8)
                    continue;
                Connection con = DriverManager.getConnection(mysqlUrl);
                String query = "SELECT * FROM " + tablesNames[i];
                Statement stmt = con.createStatement();
                ResultSet rs = stmt.executeQuery(query);
                if(rs!=null){
                    while(rs.next()) {
                        results += rs.getString(columnNames[i] + delimiter);
                    }
                }
            }
        }
    }
}
```

Listing D.4: Refactoring Task B_i^* (with annotations)

```java
1  package tasks;
2
3  import ac.collaborative.refactoring.annotations.MovableCode;
4  import java.io.IOException;
5  import java.sql.*;
6
7  public class Task_B {
8
9      public String getResults() {
10         return results;
11     }
12
13     public void setResults(String results) {
14         this.results = results;
15     }
16
17     String results = "";
18
19     @MovableCode(Description = "Reads all values from a column with a given name")
20     public void foo(String[] columnNames, String[] tablesNames, String
           ↪ mysqlUrl, String delimiter) throws IOException, SQLException {
21         if (tablesNames.length == columnNames.length) {
22             for (int i = 0; i < columnNames.length; i++) {
23                 if ((columnNames[i].contains("Demo")) ||
                       ↪ (tablesNames[i].contains("Demo"))) continue;
24                 String[] words = columnNames[i].split("\\s+");
25                 if (words.length > 8)
26                     continue;
27                 /*@MovableBegin*/
28                 Connection con = DriverManager.getConnection(mysqlUrl);
29                 String query = "SELECT * FROM " + tablesNames[i];
30                 Statement stmt = con.createStatement();
31                 ResultSet rs = stmt.executeQuery(query);
32                 String result = "";
33                 if(rs!=null){
34                     while(rs.next()) {
35                         result += rs.getString(columnNames[i] + delimiter);
36                     }
37                 }
38                 /*@MovableEnd*/
39                 results += result;
40             }
41         }
42     }
43 }
```

Table D.1: Comparison of challenges

Stage	Challenges in the code in List. D.1	Challenges in the code in List. D.3
Identifying the code to be extracted.	1. Understanding that the method runs a lot of commands in a loop, whereas we want code that runs a single command. 2. Understanding that the first six lines of the **for** loop contain checks that are relevant only to the `foo` method, and not to the general case of running a command.	1. Understanding that the method reads column values from many tables in a loop, whereas we want code that reads column values from a single table. 2. Understanding that the first five lines of the **for** loop contain checks that are relevant only to the `foo` method, and not to the general case of reading of values.
Separation of the reusable code from the context of the source class.	Replace the assignment to the instance variable with an auxiliary variable and update the instance variable outside of the reusable code.	Replace the assignment to the instance variable with an auxiliary variable and update the instance variable outside of the reusable code.

Table D.1 shows a comparison of the challenges that arise if we perform EM + MM on the reusable code fragments in List. D.1 and List. D.3.

D.2 Details of the Results

In this section, we will list the errors made when developers, who were not familiar with the code, performed EM + MM on code in a regular (unannotated) version.

1. Incorrect identification of the fragment, that caused a change in behavior in the original code.
2. The extracted fragment did not exactly perform the desired action.
3. Unnecessary commands that are not required in the general case were extracted.
4. Incorrect update of the conditions remaining in the original code. Making this change is required due to necessary changes made to the extracted code.
5. Inability to complete the task.
6. Missing handling of an edge case that was in the original code, but was lost during the extraction.
7. Unnecessary parameters were passed to the new method.
8. After introducing the use of an auxiliary variable, forget to update the original instance variable.

Errors 1, 5, 6, 7, and 8 occurred once. Errors 2, 3, and 4 occurred twice.

In relation to tasks $\langle A_i; B_i \rangle$ presented in App. D.1, during the eight times EM + MM was performed on the code of the tasks A_i or B_i, three errors were made.

1. The line of code *"if (words.length > 2) continue;"* was also extracted. This check is not required in the general case.
2. Inability to complete the task.
3. The developer forgot to update the `results` instance variable, as it was in the original code.

Bibliography

[1] Dov Fraivert and David H. Lorenz. Language support for refactorability decay prevention. In *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE '22, pages 122–134, Auckland, New Zealand, December 2022. ACM.

[2] Dov Fraivert and David H. Lorenz. Explicit code reuse recommendation. In *Companion Proceedings of the ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH '22, pages 9–10, Auckland, New Zealand, December 2022. ACM.

[3] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1992.

[4] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, February 2004.

[5] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA, 1999. With contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts.

[6] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14, September 2015.

[7] Mathieu Nassif and Martin P. Robillard. Revisiting turnover-induced knowledge loss in software projects. In *Proceedings of the 33rd IEEE International Conference on*

Software Maintenance and Evolution, ICSME '17, pages 261–272, Shanghai, China, September 2017. IEEE.

[8] Gábor Szőke. *Fighting Software Erosion with Automated Refactoring.* PhD thesis, Department of Software Engineering, University of Szeged, Szeged, Hungary, 2019.

[9] Danny Dig. The landscape of refactoring research in the last decade (keynote). In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE '17, Vancouver, BC, Canada, October 2017. ACM.

[10] Reid Holmes. *Pragmatic Software Reuse.* University of Calgary, Faculty of Graduate Studies, Calgary, Alberta, Canada, 2008.

[11] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Signature Series (Fowler). Addison-Wesley, Boston, MA, US, 2 edition, 2018. With contribution by Kent Beck.

[12] Robert C. Martin, James Newkirk, and Robert S. Koss. *Agile Software Development: Principles, Patterns, and Practices*, volume 2. Prentice-Hall, Upper Saddle River, NJ, 2003.

[13] Steve McConnell. *Code Complete.* Pearson Education, Microsoft Press, London, England, UK, 2004.

[14] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, Boston, MA, US, 2000.

[15] Reid Holmes and Robert J. Walker. Systematizing pragmatic software reuse. *ACM Transactions on Software Engineering Methodology*, 21(4):1–44, February 2013.

[16] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367, 2009.

[17] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84(10):1757–1782, October 2011.

[18] Emerson Murphy-Hill and Andrew P. Black. Breaking the barriers to successful refactoring: Observations and tools for extract method. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 421–430, Leipzig, Germany, May 2008. ACM.

[19] Olaf Seng, Johannes Stammel, and David Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, page 1909–1916, Seattle, Washington, USA, 2006. ACM.

[20] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. When does a refactoring induce bugs? an empirical study. In *Proceedings of the IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, SCAM '12, page 104–113, USA, 2012. IEEE Computer Society.

[21] Massimiliano Di Penta, Gabriele Bavota, and Fiorella Zampetti. On the relationship between refactoring actions and bugs: A differentiated replication. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 556–567, Virtual Event, USA, 2020. ACM.

[22] Donald B. Roberts. *Practical Analysis for Refactoring.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, April 1999.

[23] Elizabeth Martin, Don R. Lyon, and Brian T. Schreiber. Designing synthetic tasks for human factors research: An application to uninhabited air vehicles. In *Proceedings of the Human Factors and Ergonomics Society 42nd Annual Meeting*, pages 123–127, Chicago, Illinois, October 1998. The Society, Santa Monica, CA.

[24] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. How developers search for code: A case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 191–201, New York, NY, USA, 2015. Association for Computing Machinery.

[25] Ameer Armaly and Collin McMillan. Pragmatic source code reuse via execution record and replay. *Journal of Software: Evolution and Process*, 28(8):642–664, 2016.

[26] Yuhao Wu, Shaowei Wang, Cor-Paul Bezemer, and Katsuro Inoue. How do developers utilize source code from stack overflow? *Empirical Software Engineering*, 24(2):637–673, 2019.

תוכן עניינים

1 מבוא 1

2 רקע 7

3 הבעיה 13

4 הגישה 29

5 תמיכה של שפת התכנות 39

6 מתודולוגיה 45

7 הערכה 55

8 דיון 61

9 סיכום 67

A מדריך למפתח 69

B מדריך למשתמש 77

C השלמות לדוגמאות 81

D משימות הערכה 89

תקציר

ריחות של קוד (code smells) הם מאפיינים המעידים על הפרה של עקרונות התכן, המשפיעים לרעה על איכות הקוד. אפילו במקרה שהקוד נכתב ללא ריחות, קיים סיכון גבוה שהריחות יווצרו במהלך חיי הפרויקט.
בשביל למנוע זאת, המפתחים של הקוד יכולים לבצע ארגון קוד מניעתי (preventive refactoring), שיקטין את הסיכון, או להשאיר את הקוד כפי שהוא ולבצע ארגון קוד מתקן (corrective refactoring) אם וכאשר הריחות יופיעו.

לכל אחת מהגישות הללו יש את היתרונות והחסרונות שלה. אולם בפועל, מצד אחד מפתחים רבים נמנעים מארגון קוד מניעתי בשלב הפיתוח. זה קורה בגלל שהצורך לא ודאי ולכן ההחזר על ההשקעה הנדרשת לא מובטח. מצד שני, גם כאשר הריחות מופיעים בהמשך חיי הפרויקט, מפתחים אחרים שפחות מכירים את הקוד, נמנעים מארגון קוד מתקן בגלל המורכבות של הפעולה בשלב הזה. כתוצאה מכך מתפספסות הזדמנויות מתבקשות לארגון קוד מחדש, מה שפוגע באיכות ובתחזוקתיות של הקוד.

בעבודה זו מתיחסים לפעולת ארגון הקוד מחדש לא כפעולה אטומית, אלא כרצף של תתי פעולות. גישה זו מאפשרת לנו לחלק את האחריות על תתי הפעולות בין המפתח המקורי של הקוד, שאחראי על הכנת הקוד לארגון מחדש, ומפתח עתידי שאחראי על הביצוע בפועל של ארגון הקוד (במקרה שהריחות הופיעו). בשביל לנהל חלוקת אחריות זו, תכננו ופיתחנו קבוצה של אנוטציות (annotations) ובדיקות שמורצות בזמן קומפילציה בעזרת מעבד אנוטציות (annotation processor) שמונעים משחיקת תוכנה (software erosion) לפגוע ביכולת לבצע ארגון קוד מחדש.

תמיכת שפת התכנות בשימור היכולת לארגן את הקוד מחדש

עבודת תזה זו הוגשה כחלק מהדרישות לקבלת תואר

"מוסמך למדעים" M.Sc. במדעי המחשב

דב פרייברט

המחקר נעשה בהנחיית פרופ' דוד לורנץ

במחלקה למתמטיקה ומדעי המחשב

האוניברסיטה הפתוחה

הוגש לסנט האו"פ

ספטמבר 2023