UiO : Department of Informatics
University of Oslo

Automated Composition of Refactorings

Implementing and evaluating a search-based Extract and Move Method refactoring

Erlend Kristiansen
Master's Thesis Spring 2014

Automated Composition of Refactorings

Erlend Kristiansen

May 2, 2014

Abstract

Complex source code impacts the cost of software maintenance in a negative way. In an object-oriented context, one class may depend on a high number of other classes, thus contributing to the complexity of a program and making changing code prone to errors. Refactoring is a means to fight such complexity.

This thesis investigates whether automated refactoring can be used to lower the coupling between classes. A search-based composite refactoring combining the primitive refactorings *Extract Method* and *Move Method* is designed as a possible solution to this problem. Case studies are conducted to evaluate the effect of executing the search-based refactoring in a large code base.

Contents

Resources

Resources regarding this master's thesis can be found by visiting http://www.mn.uio.no/ifi/english/research/groups/pma/completedmasters/2014/kristiansen/. This includes program source code and installation instructions.

Acknowledgements

First and foremost, I would like to thank my supervisor Volker Stolz for guiding me through the darkest times of this thesis work.

I would also like to thank the "rancid"-guys in the 8th floor for their moral support and academic discussions.

Thanks to my mom and dad for their never-ending encouragement and support.

My sincerest gratitude goes to my girlfriend Jenny, for her love and support, for cleaning my underwear and proofreading of this thesis.

Chapter 1

Introduction

1.1 Motivation and structure

For large software projects, complex program source code is an issue. It impacts the cost of maintenance in a negative way [Ban+93], and often stalls the implementation of new functionality and other program changes. The code may be difficult to understand, the changes may introduce new bugs that are hard to find and its complexity can simply keep people from doing code changes, in fear of breaking some dependent piece of code. All these problems are related, and often lead to a vicious circle that slowly degrades the overall quality of a project.

More specifically, and in an object-oriented context, a class may depend on a number of other classes. Sometimes these intimate relationships are appropriate, and sometimes they are not. Inappropriate *coupling* between classes can make it difficult to know whether or not a change that is aimed at fixing a specific problem also alters the behavior of another part of a program.

One of the tools that are used to fight complexity and coupling in program source code is *refactoring*. The intention for this master's thesis is therefore to create an automated composite refactoring that reduces coupling between classes. The refactoring shall be able to operate automatically in all phases of a refactoring, from performing analysis to executing changes. It is also a requirement that it should be able to process large quantities of source code in a reasonable amount of time.

The current chapter proceeds in section 1.2 by describing what refactoring is. Then the project is presented in section 1.3, before the chapter is concluded with a brief discussion of related work in section 1.4.

Chapter 2 shows the workings of our refactoring together with an example illustrating how it works for a specific case. Chapter 3 presents some of the APIs and frameworks that are relevant for source code analysis and change, and that are available when using the Eclipse Platform with the Java development tools installed. Chapter 4 contains some implementation details of what was presented in chapter 2. Chapter 5 presents a couple of case studies used to evaluate several aspects of our refactoring. The whole thesis is then winded up in chapter 6 with conclusions and future work.

1.2 What is refactoring?

This question is best answered by first defining the concept of a *refactoring*, what it is to *refactor*, and then discuss what aspects of programming make people want to refactor their code.

1.2.1 Defining refactoring

Martin Fowler, in his classic book on refactoring [Fow99], defines a refactoring like this:

> *Refactoring* (noun): a change made to the internal structure[1] of software to make it easier to understand and cheaper to modify without changing its observable behavior. [Fow99, p. 53]

This definition assigns additional meaning to the word *refactoring*, beyond the composition of the prefix *re-*, usually meaning something like "again" or "anew", and the word *factoring*, which can mean to isolate the *factors* of something. Here a *factor* would be close to the mathematical definition of something that divides a quantity, without leaving a remainder. Fowler is mixing the *motivation* behind refactoring into his definition. Instead it could be more refined, formed to only consider the *mechanical* and *behavioral* aspects of refactoring. That is to factor the program again, putting it together in a different way than before, while preserving the behavior of the program. An alternative definition could then be:

Definition. A *refactoring* is a transformation done to a program without altering its external behavior.

From this we can conclude that a refactoring primarily changes how the *code* of a program is perceived by the *programmer*, and not the *behavior* experienced by any user of the program. Although the logical meaning is preserved, such changes could potentially alter the program's behavior when it comes to performance gain or -penalties. So any logic depending on the performance of a program could make the program behave differently after a refactoring.

In the extreme case one could argue that software obfuscation is refactoring. It is often used to protect proprietary software. It restrains uninvited viewers, so they have a hard time analyzing code that they are not supposed to know how works. This could be a problem when using a language that is possible to decompile, such as Java.

Obfuscation could be done composing many, more or less randomly chosen, refactorings. Then the question arises whether it can be called a *composite refactoring* or not (see section 1.2.9 on page 22)? The answer is not obvious. First, there is no way to describe the mechanics of software obfuscation, because there are infinitely many ways to do that. Second, obfuscation can be thought of as *one operation*: Either the code is

[1]The structure observable by the programmer.

obfuscated, or it is not. Third, it makes no sense to call software obfuscation *a refactoring*, since it holds different meaning to different people.

This last point is important, since one of the motivations behind defining different refactorings, is to establish a *vocabulary* for software professionals to use when reasoning about and discussing programs, similar to the motivation behind design patterns [Gam+95].

1.2.2 The etymology of 'refactoring'

It is a little difficult to pinpoint the exact origin of the word "refactoring", as it seems to have evolved as part of a colloquial terminology, more than a scientific term. There is no authoritative source for a formal definition of it.

According to Martin Fowler [Fow03], there may also be more than one origin of the word. The most well-known source, when it comes to the origin of *refactoring*, is the Smalltalk community and their infamous Refactoring Browser described in the article *A Refactoring Tool for Smalltalk* [RBJ97], published in 1997. Allegedly [Fow03], the metaphor of factoring programs was also present in the Forth community, and the word "refactoring" is mentioned in a book by Leo Brodie, called *Thinking Forth* [Bro04], first published in 1984[1]. The exact word is only printed one place [Bro04, p. 232], but the term *factoring* is prominent in the book, which also contains a whole chapter dedicated to (re)factoring, and how to keep the (Forth) code clean and maintainable.

> . . . good factoring technique is perhaps the most important skill for a Forth programmer. [Bro04, p. 172]

Brodie also express what *factoring* means to him:

> Factoring means organizing code into useful fragments. To make a fragment useful, you often must separate reusable parts from non-reusable parts. The reusable parts become new definitions. The non-reusable parts become arguments or parameters to the definitions. [Bro04, p. 172]

Fowler claims that the usage of the word *refactoring* did not pass between the Forth and Smalltalk communities, but that it emerged independently in each of the communities.

1.2.3 Reasons for refactoring

There are many reasons why people want to refactor their programs. They can for instance do it to remove duplication, break up long methods or to introduce design patterns into their software systems. The shared trait for

[1] *Thinking Forth* was first published in 1984 by the Forth Interest Group. Then it was reprinted in 1994 with minor typographical corrections, before it was transcribed into an electronic edition typeset in LaTeX and published under a Creative Commons license in 2004. The edition cited here is the 2004 edition, but the content should essentially be as in 1984.

all these are that peoples' intentions are to make their programs *better*, in some sense. But what aspects of their programs are becoming improved?

As just mentioned, people often refactor to get rid of duplication. They are moving identical or similar code into methods, and are pushing methods up or down in their class hierarchies. They are making template methods for overlapping algorithms/functionality, and so on. It is all about gathering what belongs together and putting it all in one place. The resulting code is then easier to maintain. When removing the implicit coupling[1] between code snippets, the location of a bug is limited to only one place, and new functionality need only to be added to this one place, instead of a number of places people might not even remember.

A problem you often encounter when programming, is that a program contains a lot of long and hard-to-grasp methods. It can then help to break the methods into smaller ones, using the *Extract Method* refactoring [Fow99]. Then you may discover something about a program that you were not aware of before; revealing bugs you did not know about or could not find due to the complex structure of your program. Making the methods smaller and giving good names to the new ones clarifies the algorithms and enhances the *understandability* of the program (see section 1.2.4 on the next page). This makes refactoring an excellent method for exploring unknown program code, or code that you had forgotten that you wrote.

Most primitive refactorings are simple, and usually involves moving code around [Ker05]. The motivation behind them may first be revealed when they are combined into larger — higher level — refactorings, called *composite refactorings* (see section 1.2.9 on page 22). Often the goal of such a series of refactorings is a design pattern. Thus the design can *evolve* throughout the lifetime of a program, as opposed to designing up-front. It is all about being structured and taking small steps to improve a program's design.

Many software design pattern are aimed at lowering the coupling between different classes and different layers of logic. One of the most famous is perhaps the *Model-View-Controller* [Gam+95] pattern. It is aimed at lowering the coupling between the user interface, the business logic and the data representation of a program. This also has the added benefit that the business logic could much easier be the target of automated tests, thus increasing the productivity in the software development process.

Another effect of refactoring is that with the increased separation of concerns coming out of many refactorings, the *performance* can be improved. When profiling programs, the problematic parts are narrowed down to smaller parts of the code, which are easier to tune, and optimization can be performed only where needed and in a more effective way [Fow99].

Last, but not least, and this should probably be the best reason to refactor, is to refactor to *facilitate a program change*. If one has managed to keep one's code clean and tidy, and the code is not bloated with design patterns that are not ever going to be needed, then some refactoring might

[1]When duplicating code, the duplicate pieces of code might not be coupled, apart from representing the same functionality. So if this functionality is going to change, it might need to change in more than one place, thus creating an implicit coupling between multiple pieces of code.

be needed to introduce a design pattern that is appropriate for the change that is going to happen.

Refactoring program code — with a goal in mind — can give the code itself more value. That is in the form of robustness to bugs, understandability and maintainability. Having robust code is an obvious advantage, but understandability and maintainability are both very important aspects of software development. By incorporating refactoring in the development process, bugs are found faster, new functionality is added more easily and code is easier to understand by the next person exposed to it, which might as well be the person who wrote it. The consequence of this, is that refactoring can increase the average productivity of the development process, and thus also add to the monetary value of a business in the long run. The perspective on productivity and money should also be able to open the eyes of the many nearsighted managers that seldom see beyond the next milestone.

1.2.4 The magical number seven

The article *The magical number seven, plus or minus two: some limits on our capacity for processing information* [Mil56] by George A. Miller, was published in the journal Psychological Review in 1956. It presents evidence that support that the capacity of the number of objects a human being can hold in its working memory is roughly seven, plus or minus two objects. This number varies a bit depending on the nature and complexity of the objects, but is according to Miller "... never changing so much as to be unrecognizable."

Miller's article culminates in the section called *Recoding*, a term he borrows from communication theory. The central result in this section is that by recoding information, the capacity of the amount of information that a human can process at a time is increased. By *recoding*, Miller means to group objects together in chunks, and give each chunk a new name that it can be remembered by.

> ... recoding is an extremely powerful weapon for increasing the amount of information that we can deal with. [Mil56, p. 95]

By organizing objects into patterns of ever growing depth, one can memorize and process a much larger amount of data than if it were to be represented as its basic pieces. This grouping and renaming is analogous to how many refactorings work, by grouping pieces of code and give them a new name. Examples are the fundamental *Extract Method* and *Extract Class* refactorings [Fow99].

An example from the article addresses the problem of memorizing a sequence of binary digits. The example presented here is a slightly modified version of the one presented in the original article [Mil56], but it preserves the essence of it. Let us say we have the following sequence of 16 binary digits: "1010001001110011". Most of us will have a hard time memorizing this sequence by only reading it once or twice. Imagine if we instead translate it to this sequence: “A273”. If you have a background from computer science, it will be obvious that the latter sequence is the first sequence recoded to be represented by digits in base 16. Most people should be able to memorize this last sequence by only looking at it once.

Another result from the Miller article is that when the amount of information a human must interpret increases, it is crucial that the translation from one code to another must be almost automatic for the subject to be able to remember the translation, before he is presented with new information to recode. Thus learning and understanding how to best organize certain kinds of data is essential to efficiently handle that kind of data in the future. This is much like when humans learn to read. First they must learn how to recognize letters. Then they can learn distinct words, and later read sequences of words that form whole sentences. Eventually, most of them will be able to read whole books and briefly retell the important parts of its content. This suggests that the use of design patterns is a good idea when reasoning about computer programs. With extensive use of design patterns when creating complex program structures, one does not always have to read whole classes of code to comprehend how they function, it may be sufficient to only see the name of a class to almost fully understand its responsibilities.

> Our language is tremendously useful for repackaging material into a few chunks rich in information. [Mil56, p. 95]

Without further evidence, these results at least indicate that refactoring source code into smaller units with higher cohesion and, when needed, introducing appropriate design patterns, should aid in the cause of creating computer programs that are easier to maintain and have code that is easier (and better) understood.

1.2.5 Notable contributions to the refactoring literature

1992 William F. Opdyke submits his doctoral dissertation called *Refactoring Object-Oriented Frameworks* [Opd92]. This work defines a set of refactorings that are behavior-preserving given that their preconditions are met. The dissertation is focused on the automation of refactorings.

1999 Martin Fowler et al.: *Refactoring: Improving the Design of Existing Code* [Fow99]. This is maybe the most influential text on refactoring. It bares similarities with Opdykes thesis [Opd92] in the way that it provides a catalog of refactorings. But Fowler’s book is more about the craft of refactoring, as he focuses on establishing a vocabulary for refactoring, together with the mechanics of different refactorings and when to perform them. His methodology is also founded on the principles of test-driven development.

2005 Joshua Kerievsky: *Refactoring to Patterns* [Ker05]. This book is heavily influenced by Fowler’s *Refactoring* [Fow99] and the “Gang of

Four" *Design Patterns* [Gam+95]. It is building on the refactoring catalogue from Fowler's book, but is trying to bridge the gap between *refactoring* and *design patterns* by providing a series of higher-level composite refactorings, that makes code evolve toward or away from certain design patterns. The book is trying to build up the reader's intuition around *why* one would want to use a particular design pattern, and not just *how*. The book is encouraging evolutionary design (see section 1.2.7 on the next page).

1.2.6 Tool support (for Java)

This section will briefly compare the refactoring support of the three IDEs Eclipse, IntelliJ IDEA[1] and NetBeans. These are the most popular Java IDEs [11].

All three IDEs provide support for the most useful refactorings, like the different extract, move and rename refactorings. In fact, Java-targeted IDEs are known for their good refactoring support, so this did not appear as a big surprise.

The IDEs seem to have excellent support for the *Extract Method* refactoring, so at least they have all passed the first "refactoring rubicon" [Fow01; VJ12].

Regarding the *Move Method* refactoring [Fow99], the Eclipse and IntelliJ IDEs do the job in very similar manners. In most situations they both do a satisfying job by producing the expected outcome. But they do nothing to check that the result does not break the semantics of the program (see section 1.2.11 on page 23). The NetBeans IDE implements this refactoring in a somewhat unsophisticated way. For starters, the refactoring's default destination for the move, is the same class as the method already resides in, although it refuses to perform the refactoring if chosen. But the worst part is, that if moving the method `f` of the class `C` to the class `X`, it will break the code. The result is shown in listing 1.

```java
public class C {
    private X x;
    ...
    public void f() {
        x.m();
        x.n();
    }
}
```

```java
public class X {
    ...
    public void f(C c) {
        c.x.m();
        c.x.n();
    }
}
```

Listing 1: Moving method `f` from `C` to `X`.

NetBeans will try to create code that call the methods `m` and `n` of `X` by accessing them through `c.x`, where `c` is a parameter of type `C` that is added the method `f` when it is moved. (This is seldom the desired outcome of this refactoring, but ironically, this "feature" keeps NetBeans from breaking the

[1]The IDE under comparison is the Community Edition.

code in the example from section 1.2.11 on page 23.) If `c.x` for some reason is inaccessible to `X`, as in this case, the refactoring breaks the code, and it will not compile. NetBeans presents a preview of the refactoring outcome, but the preview does not catch it if the IDE is about break the program.

The IDEs under investigation seem to have fairly good support for primitive refactorings, but what about more complex ones, such as *Extract Class* [Fow99]? IntelliJ handles this in a fairly good manner, although, in the case of private methods, it leaves unused methods behind. These are methods that delegate to a field with the type of the new class, but are not used anywhere. Eclipse has added its own quirk to the *Extract Class* refactoring, and only allows for *fields* to be moved to a new class, *not methods.* This makes it effectively only extracting a data structure, and calling it *Extract Class* is a little misleading. One would often be better off with textual extract and paste than using the *Extract Class* refactoring in Eclipse. When it comes to NetBeans, it does not even show an attempt on providing this refactoring.

1.2.7 The relation to design patterns

Refactoring and design patterns have at least one thing in common, they are both promoted by advocates of *clean code* [MC09] as fundamental tools on the road to more maintainable and extendable source code.

> Design patterns help you determine how to reorganize a design, and they can reduce the amount of refactoring you need to do later. [Gam+95, p. 353]

Although sometimes associated with over-engineering [Ker05; Fow99], design patterns are in general assumed to be good for maintainability of source code. That may be because many of them are designed to support the *open/closed principle* of object-oriented programming. The principle was first formulated by Bertrand Meyer, the creator of the Eiffel programming language, like this: "Modules should be both open and closed." [Mey88] It has been popularized, with this as a common version:

> Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

Maintainability is often thought of as the ability to be able to introduce new functionality without having to change too much of the old code. When refactoring, the motivation is often to facilitate adding new functionality. It is about factoring the old code in a way that makes the new functionality being able to benefit from the functionality already residing in a software system, without having to copy old code into new. Then, next time someone shall add new functionality, it is less likely that the old code has to change. Assuming that a design pattern is the best way to get rid of duplication and assist in implementing new functionality, it is reasonable to conclude that a design pattern often is the target of a series of refactorings. Having a repertoire of design patterns can also help in knowing when and how to refactor a program to make it reflect certain desired characteristics.

> There is a natural relation between patterns and refactorings. Patterns are where you want to be; refactorings are ways to get there from somewhere else. [Fow99, p. 107]

This quote is wise in many contexts, but it is not always appropriate to say "Patterns are where you want to be...". *Sometimes*, patterns are where you want to be, but only because it will benefit your design. It is not true that one should always try to incorporate as many design patterns as possible into a program. It is not like they have intrinsic value. They only add value to a system when they support its design. Otherwise, the use of design patterns may only lead to a program that is more complex than necessary.

> The overuse of patterns tends to result from being patterns happy. We are *patterns happy* when we become so enamored of patterns that we simply must use them in our code. [Ker05, p. 24]

This can easily happen when relying largely on up-front design. Then it is natural, in the very beginning, to try to build in all the flexibility that one believes will be necessary throughout the lifetime of a software system. According to Joshua Kerievsky "That sounds reasonable — if you happen to be psychic." [Ker05, p. 1] He is advocating what he believes is a better approach: To let software continually evolve. To start with a simple design that meets today's needs, and tackle future needs by refactoring to satisfy them. He believes that this is a more economic approach than investing time and money into a design that inevitably is going to change. By relying on continuously refactoring a system, its design can be made simpler without sacrificing flexibility. To be able to fully rely on this approach, it is of utter importance to have a reliable suit of tests to lean on (see section 1.2.12 on page 25). This makes the design process more natural and less characterized by difficult decisions that has to be made before proceeding in the process, and that is going to define a project for all of its unforeseeable future.

1.2.8 The impact on software quality

What is software quality?

The term *software quality* has many meanings. It all depends on the context we put it in. If we look at it with the eyes of a software developer, it usually means that the software is easily maintainable and testable, or in other words, that it is *well designed*. This often correlates with the management scale, where *keeping the schedule* and *customer satisfaction* is at the center. From the customers point of view, in addition to good usability, *performance* and *lack of bugs* is always appreciated, measurements that are also shared by the software developer. (In addition, such things as good documentation could be measured, but this is out of the scope of this document.)

The impact on performance

> Refactoring certainly will make software go more slowly[1], but it also makes the software more amenable to performance tuning. [Fow99, p. 69]

There is a common belief that refactoring compromises performance, due to increased degree of indirection and that polymorphism is slower than conditionals.

In a survey, Demeyer [Dem02] disproves this view in the case of polymorphism. He did an experiment on, what he calls, "Transform Self Type Checks" where you introduce a new polymorphic method and a new class hierarchy to get rid of a class' type checking of a "type attribute". He uses this kind of transformation to represent other ways of replacing conditionals with polymorphism as well. The experiment is performed on the C++ programming language and with three different compilers and platforms. Demeyer concludes that, with compiler optimization turned on, polymorphism beats middle to large sized if-statements and does as well as case-statements. (In accordance with his hypothesis, due to similarities between the way C++ handles polymorphism and case-statements.)

> The interesting thing about performance is that if you analyze most programs, you find that they waste most of their time in a small fraction of the code. [Fow99, p. 70]

So, although an increased amount of method calls could potentially slow down programs, one should avoid premature optimization and sacrificing good design, leaving the performance tuning until after profiling the software and having isolated the actual problem areas.

1.2.9 Composite refactorings

Generally, when thinking about refactoring, at the mechanical level, there are essentially two kinds of refactorings. There are the *primitive* refactorings, and the *composite* refactorings.

Definition. A *primitive refactoring* is a refactoring that cannot be expressed in terms of other refactorings.

Examples are the *Pull Up Field* and *Pull Up Method* refactorings [Fow99], that move members up in their class hierarchies.

Definition. A *composite refactoring* is a refactoring that can be expressed in terms of two or more other refactorings.

An example of a composite refactoring is the *Extract Superclass* refactoring [Fow99]. In its simplest form, it is composed of the previously described primitive refactorings, in addition to the *Pull Up Constructor Body* refactoring [Fow99]. It works by creating an abstract superclass that the target

[1]With today's compiler optimization techniques and performance tuning of e.g. the Java virtual machine, the penalties of object creation and method calls are debatable.

class(es) inherits from, then by applying *Pull Up Field*, *Pull Up Method* and *Pull Up Constructor Body* on the members that are to be members of the new superclass. If there are multiple classes in play, their interfaces may need to be united with the help of some rename refactorings, before extracting the superclass. For an overview of the *Extract Superclass* refactoring, see figure 1.1.

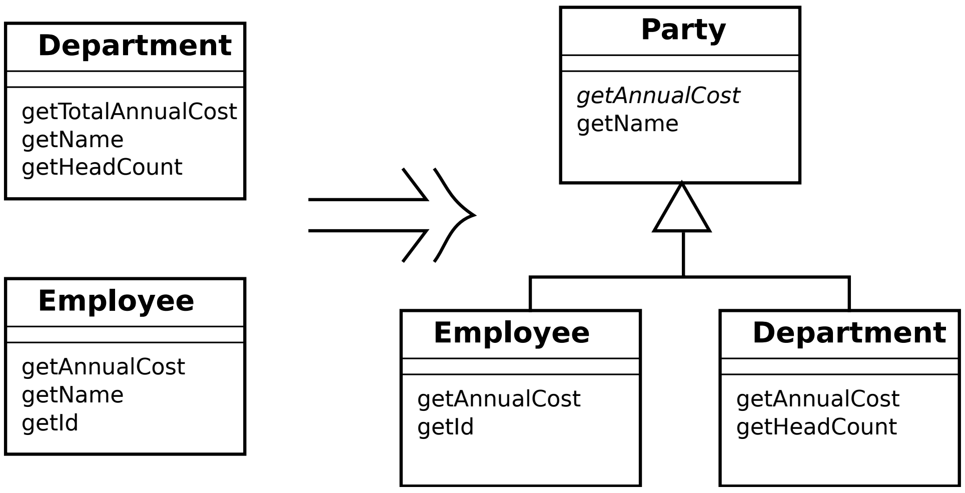

Figure 1.1: The *Extract Superclass* refactoring, with united interfaces. (Taken from http://refactoring.com/catalog/extractSuperclass.html.)

1.2.10 Manual vs. automated refactorings

Refactoring is something every programmer does, even if she does not know the term *refactoring*. Every refinement of source code that does not alter the program's behavior is a refactoring. For small refactorings, such as *Extract Method*, executing it manually is a manageable task, but is still prone to errors. Getting it right the first time is not easy, considering the method signature and all the other aspects of the refactoring that has to be in place.

Consider the renaming of classes, methods and fields. For complex programs these refactorings are almost impossible to get right. Attacking them with textual search and replace, or even regular expressions, will fall short on these tasks. Then it is crucial to have proper tool support that can perform them automatically. Tools that can parse source code and thus have semantic knowledge about which occurrences of which names belong to what construct in the program. For even trying to perform one of these complex tasks manually, one would have to be very confident on the existing test suite (see section 1.2.12 on page 25).

1.2.11 Correctness of refactorings

For automated refactorings to be truly useful, they must show a high degree of behavior preservation. This last sentence might seem obvious, but there are examples of refactorings in existing tools that break programs. In an ideal world, every automated refactoring would be “complete”, in the sense that it would never break a program. In an ideal world, every program would also be free from bugs. In modern IDEs the implemented automated refactorings are working for *most* cases, which is enough for making them useful.

I will now present an example of a *corner case* where a program breaks when a refactoring is applied. The example shows an *Extract Method* refactoring followed by a *Move Method* refactoring that breaks a program in both the Eclipse and IntelliJ IDEs[1]. The target and the destination for the composed refactoring are shown in listing 2. Note that the method `m(C c)` of class `X` assigns to the field `x` of the argument `c` that has type `C`.

Refactoring target

```
1 public class C {
2   public X x = new X();
3
4   public void f() {
5     x.m(this);
6     // Not the same x.
7     x.n();
8   }
9 }
```

Method destination

```
1 public class X {
2   public void m(C c) {
3     c.x = new X();
4     // If m is called from
5     // c, then c.x no longer
6     // equals 'this'.
7   }
8   public void n() {}
9 }
```

Listing 2: The target and the destination for the composition of the *Extract Method* and *Move Method* refactorings.

The refactoring sequence works by extracting line 6 through 8 from the original class `C` into a method `f` with the statements from those lines as its method body (but with the comment left out, since it will no longer hold any meaning). The method is then moved to the class `X`. The result is shown in listing 3.

Before the refactoring, the methods `m` and `n` of class `X` are called on different object instances (see line 6 and 8 of the original class `C` in listing 2). After the refactoring, they are called on the same object, and the statement on line 3 of class `X` (in listing 3) no longer has the desired effect in our example. The method `f` of class `C` is now calling the method `f` of class `X` (see line 5 of class `C` in listing 3), and the program now behaves different than before.

The bug introduced in the previous example is of such a nature[2] that it is very difficult to spot if the refactored code is not covered by tests. It does not generate compilation errors, and will thus only result in a runtime error or corrupted data, which might be hard to detect.

[1]The NetBeans IDE handles this particular situation without altering the program's behavior, mainly because its *Move Method* refactoring implementation is a bit flawed in other ways (see section 1.2.6 on page 19).

[2]Caused by aliasing.

```java
public class C {
    public X x = new X();

    public void f() {
        x.f(this);
    }
}
```

```java
public class X {
    public void m(C c) {
        c.x = new X();
    }
    public void n() {}
    // Extracted and
    // moved method.
    public void f(C c) {
        m(c);
        n();
    }
}
```

Listing 3: The result of the composed refactoring.

1.2.12 Refactoring and the importance of testing

> If you want to refactor, the essential precondition is having solid tests. [Fow99]

When refactoring, there are roughly three classes of errors that can be made. The first class of errors is the one that makes the code unable to compile. These *compile-time* errors are of the nicer kind. They flash up at the moment they are made (at least when using an IDE), and are usually easy to fix. The second class is the *runtime* errors. Although these errors take a bit longer to surface, they usually manifest after some time in an illegal argument exception, null pointer exception or similar during the program execution. These kinds of errors are a bit harder to handle, but at least they will show, eventually. Then there are the *behavior-changing* errors. These errors are of the worst kind. They do not show up during compilation and they do not turn on a blinking red light during runtime either. The program can seem to work perfectly fine with them in play, but the business logic can be damaged in ways that will only show up over time.

For discovering runtime errors and behavior changes when refactoring, it is essential to have good test coverage. Testing in this context means writing automated tests. Manual testing may have its uses, but when refactoring, it is automated unit testing that dominate. For discovering behavior changes it is especially important to have tests that cover potential problems, since these kinds of errors do not reveal themselves.

Unit testing is not a way to *prove* that a program is correct, but it is a way to make you confident that it *probably* works as desired. In the context of test-driven development (commonly known as TDD), the tests are even a way to define how the program is *supposed* to work. It is then, by definition, working if the tests are passing.

If the test coverage for a code base is perfect, then it should, theoretically, be risk-free to perform refactorings on it. This is why automated tests and refactoring is such a great match.

Testing the code from correctness section

The worst thing that can happen when refactoring is to introduce changes to the behavior of a program, as in the example on section 1.2.11 on page 23. This example may be trivial, but the essence is clear. The only problem with the example is that it is not clear how to create automated tests for it, without changing it in intrusive ways.

Unit tests, as they are known from the different xUnit frameworks around, are only suitable to test the *result* of isolated operations. They can not easily (if at all) observe the *history* of a program.

This problem is still open.

1.3 The project

In this section we look at the work that will be done for this project, its building blocks, pose some research questions and present some of the methodologies used.

1.3.1 Project description

The aim of this master's project will be to explore the relationship between the *Extract Method* and the *Move Method* refactorings. This will be done by composing the two into a composite refactoring. The refactoring will be called the *Extract and Move Method* refactoring.

The two primitive refactorings *Extract Method* and *Move Method* must already be implemented in a tool, so the *Extract and Move Method* refactoring can be built on top of these.

The composition of the *Extract Method* and *Move Method* refactorings springs naturally out of the need to move procedures closer to the data they manipulate. This composed refactoring is not well described in the literature, but it is implemented in at least one tool called CodeRush, which is an extension for MS Visual Studio. In CodeRush it is called *Extract Method to Type*[1], but I choose to call it *Extract and Move Method*, since I feel this better communicates which primitive refactorings it is composed of.

The project will consist of implementing the *Extract and Move Method* refactoring, as well as executing it over a larger code base, as a case study. To be able to execute the refactoring automatically, I have to make it analyze code to determine the best selections to extract into new methods.

1.3.2 The premises

Before we can start manipulating source code, and write a tool for doing so, we need to decide on a programming language for the code we are going to manipulate. Also, since we do not want to start from scratch by implementing primitive refactorings ourselves, we need to choose an existing tool that provides the needed refactorings. In addition to be able to perform

[1] https://help.devexpress.com/#CodeRush/CustomDocument6710

changes, we need a framework for analyzing source code for the language we select.

Choosing the target language

Choosing which programming language the code that will be manipulated shall be written in, is not a very difficult task. We choose to limit the possible languages to the object-oriented programming languages, since most of the terminology and literature regarding refactoring comes from the world of object-oriented programming. In addition, the language must have existing tool support for refactoring.

The Java programming language is the dominating language when it comes to example code in the literature of refactoring, and is thus a natural choice. Java is perhaps the most influential programming language in the world today, with its Java Virtual Machine that runs on all of the most popular computer architectures and also supports dozens of other programming languages[1], with Scala, Clojure and Groovy as the most prominent ones. Java is currently the language that every other programming language is compared against. It is also the primary programming language for the author of this thesis.

Choosing the tools

When choosing a tool for manipulating Java, there are certain criteria that have to be met. First of all, the tool should have some existing refactoring support that this thesis can build upon. Secondly, it should provide some kind of framework for parsing and analyzing Java source code. Third, it should itself be open source. This is both because of the need to be able to browse the code for the existing refactorings that is contained in the tool, and also because open source projects hold value in them selves. Another important aspect to consider, is that open source projects of a certain size usually has large communities of people connected to them, which are committed to answering questions regarding the use and misuse of the products, that to a large degree is made by the community itself.

There is a certain class of tools that meet these criteria, namely the class of *IDEs*[2]. These are programs that are meant to support the whole production cycle of a computer program, and the most popular IDEs that support Java generally have quite good refactoring support.

The main contenders for this thesis are the Eclipse IDE, with the Java development tools (JDT), the IntelliJ IDEA Community Edition and the NetBeans IDE (see section 1.2.6 on page 19). Eclipse and NetBeans are both free, open source and community driven, while the IntelliJ IDEA has an open-sourced community edition that is free of charge, but also offers an Ultimate Edition with an extended set of features, at additional cost. All three IDEs supports adding plugins to extend their functionality and tools that can be used to parse and analyze Java source code. But one of the

[1]They compile to Java bytecode.

[2]*Integrated Development Environment*

IDEs stand out as a favorite, and that is the Eclipse IDE. This is the most popular [11] among the three and seems to be de facto standard IDE for Java development regardless of platform.

1.3.3 The primitive refactorings

The refactorings presented here are the primitive refactorings used in this project. They are the abstract building blocks used by the *Extract and Move Method* refactoring.

The Extract Method refactoring. The *Extract Method* refactoring [Fow99] is used to extract a fragment of code from its context and into a new method. A call to the new method is inlined where the fragment was before. It is used to break code into logical units, with names that explain their purpose.

An example of an *Extract Method* refactoring is shown in listing 4. It shows a method containing calls to the methods **foo** and **bar** of a type **X**. These statements are then extracted into the new method **fooBar**.

```
Before
class C {
  void method() {
    X x = new X();
    x.foo(); x.bar();
  }
}
```

```
After
class C {
  void method() {
    X x = new X();
    fooBar(x);
  }
  void fooBar(X x) {
    x.foo(); x.bar();
  }
}
```

Listing 4: An example of an *Extract Method* refactoring.

The Move Method refactoring. The *Move Method* refactoring [Fow99] is used to move a method from one class to another. This can be appropriate if the method is using more features of another class than of the class in which it is currently defined.

Listing 5 on the next page shows an example of this refactoring. Here, the method **fooBar** is moved from class **C** to class **X**.

1.3.4 The Extract and Move Method refactoring

The *Extract and Move Method* refactoring is a composite refactoring composed of the primitive refactorings *Extract Method* and *Move Method*. The effect of this refactoring on source code is the same as when extracting a method and moving it to another class. Conceptually, this is done without

```
Before
class C {
  void method() {
    X x = new X();
    fooBar(x);
  }
  void fooBar(X x) {
    x.foo(); x.bar();
  }
}

class X {
  void foo(){/*...*/}
  void bar(){/*...*/}
}
```

```
After
class C {
  void method() {
    X x = new X();
    x.fooBar();
  }
}

class X {
  void foo(){/*...*/}
  void bar(){/*...*/}
  void fooBar() {
    foo(); bar();
  }
}
```

Listing 5: An example of a *Move Method* refactoring.

an intermediate step. In practice, as we shall see later, an intermediate step may be necessary.

An example of this composite refactoring is shown in listing 6 on the following page. The example joins the examples from listing 4 and listing 5. This means that the selection consisting of the consecutive calls to the methods `foo` and `bar`, is extracted into a new method `fooBar` located in the class `X`.

1.3.5 The Coupling Between Object Classes metric

The best known metric for measuring coupling between classes in object-oriented software is called *Coupling Between Object Classes*, usually abbreviated as CBO. The metric is defined in the article *A Metrics Suite for Object Oriented Design* [CK94] by Chidamber and Kemerer, published in 1994.

Definition. *CBO* for a class is a count of the number of other classes to which it is coupled.

An object is coupled to another object if one of them acts on the other by using methods or instance variables of the other object. This relation goes both ways, so both outgoing and incoming uses are counted. Each coupling relationship is only considered once when measuring CBO for a class.

How can the Extract and Move Method refactoring improve CBO? Listing 7 on page 35 shows how CBO changes for a class when it is refactored with the *Extract and Move Method* refactoring. In the example we consider only the CBO value of class `C`.

```
Before
class C {
  void method() {
    X x = new X();
    x.foo(); x.bar();
  }
}

class X {
  void foo(){/*...*/}
  void bar(){/*...*/}
}
```

```
After
class C {
  void method() {
    X x = new X();
    x.fooBar();
  }
}

class X {
  void foo(){/*...*/}
  void bar(){/*...*/}
  void fooBar() {
    foo(); bar();
  }
}
```

Listing 6: An example of an *Extract and Move Method* refactoring.

Before refactoring the class **C** with the *Extract and Move Method* refactoring, it has a CBO value of 4. The class uses members of the classes **A** and **B**, which accounts for 2 of the coupling relationships of class **C**. In addition to this, it uses its variable **x** with type **X** and also the methods **foo** and **bar** declared in class **Y**, giving it a total CBO value of 4.

The after-part of the example code in listing 7 shows the result of extracting the lines 5 and 6 of class **C** into a new method **fooBar**, with a subsequent move of it to class **X**.

With respect to the CBO metric, the refactoring action accomplishes something important: It eliminates the uses of class **Y** from class **C**. This means that the class **C** is no longer coupled to **Y**, only the classes **A**, **B** and **X**. The CBO value of class **C** is therefore 3 after the refactoring, while no other class have received any increase in CBO.

The example shown here is an ideal situation. Coupling is reduced for one class without any increase of coupling for another class. There is also another important point: It is the fact that to reduce the CBO value for a class, we need to remove *all* its uses of another class. This is done for the class **C** in listing 7 on page 35, where all uses of class **Y** is removed by the *Extract and Move Method* refactoring.

1.3.6 Research questions

The main question that I seek an answer to in this thesis is:

> Is it possible to automate the analysis and execution of the *Extract and Move Method* refactoring, and do so for all of the code of a larger project?

The secondary questions will then be:

Can we do this efficiently? Can we automate the analysis and execution of the refactoring so it can be run in a reasonable amount of time?

Can we perform changes safely? Can we take actions to prevent the refactoring from breaking code? By “breaking code” we mean to either do changes that do not compile, or make changes that alter the behavior of a program.

Can we improve the quality of source code? Assuming that the refactoring is safe: Is it feasible to assure that code we refactor actually gets better in terms of coupling?

How can the automation of the refactoring be helpful? Assuming the refactoring does in fact improve the quality of source code and is safe to use: What is the usefulness of the refactoring in a software development setting? In what parts of the development process can the refactoring play a role?

1.3.7 Methodology

This section will present some of the methodologies used during the work of this thesis.

Evolutionary design

In the programming work for this project, I have tried using a design strategy called evolutionary design, also known as continuous or incremental design [Sho04]. It is a software design strategy advocated by the Extreme Programming community. The essence of the strategy is that you should let the design of your program evolve naturally as your requirements change. This is seen in contrast with up-front design, where design decisions are made early in the process.

The motivation behind evolutionary design is to keep the design of software as simple as possible. This means not introducing unneeded functionality into a program. You should defer introducing flexibility into your software, until it is needed to be able to add functionality in a clean way.

Holding up design decisions, implies that the time will eventually come when decisions have to be made. The flexibility of the design then relies on the programmer’s abilities to perform the necessary refactoring, and her confidence in those abilities. From my experience working on this project, I can say that this confidence is greatly enhanced by having automated tests to rely on (see section 1.3.7 on the next page).

The choice of going for evolutionary design developed naturally. As Fowler points out in his article *Is Design Dead?*, evolutionary design much resembles the “code and fix” development strategy [Fow04]. A strategy which most of us have practiced in school. This was also the case when I first started this work. I had to learn the inner workings of Eclipse and its refactoring-related plugins. That meant a lot of fumbling around with code I did not know, in a trial and error fashion. Eventually I started writing tests for my code, and my design began to evolve.

Test-driven development

As mentioned before, the project started out as a classic code and fix development process. My focus was aimed at getting something to work, rather than doing so according to best practice. This resulted in a project that got out of its starting blocks, but it was not accompanied by any tests. Hence it was soon difficult to make any code changes with the confidence that the program was still correct afterwards (assuming it was so before changing it). I always knew that I had to introduce some tests at one point, but this experience accelerated the process of leading me onto the path of testing.

I then wrote tests for the core functionality of the plugin, and thus gained more confidence in the correctness of my code. I could now perform quite drastic changes without "wetting my pants". After this, nearly all of the semantic changes done to the business logic of the project, or the addition of new functionality, were made in a test-driven manner. This means that before performing any changes, I would define the desired functionality through a set of tests. I would then run the tests to check that they were run and that they did not pass. Then I would do any code changes necessary to make the tests pass. The definition of how the program is supposed to operate is then captured by the tests. However, this does not prove the correctness of the analysis leading to the test definitions.

Case studies

The case study methodology is used to show how the *Extract and Move Method* refactoring performs on real code, not just toy examples. The case studies are used to analyze our project so we can conclude on its completeness and usefulness.

Dogfooding

Dogfooding is a methodology where you use your own tools to do your job, also referred to as "eating your own dog food" [Har06]. It is used in this project to see if we can refactor our own refactoring code and still use it to refactor other code.

1.4 Related work

Here, some work is presented that relate to automated composition of refactorings.

1.4.1 Search-based refactoring

Search-Based Refactoring: an empirical study [OC08] is a paper by Mark O'Keeffe and Mel Ó Cinnéide published in 2008. The authors present an empirical study of different algorithmic approaches to search-based refactoring.

The common approach for all these algorithms is to generate a set of changes to a program for then to use a "fitness function" to evaluate if they improve its design or not. The fitness function consists of a weighted sum of different object-oriented metrics.

Among other things, the authors conclude that even with small input programs, their solution representation is memory-intensive, at least for some algorithms. The programs they refactor on have in average 4,000 lines of code, spread over 57 classes. I.e. considerably smaller than one of the programs that will be subject to refactoring in this project.

1.4.2 "Making Program Refactoring Safer"

This is the name of an article [Soa+10] about providing a way to improve safety during refactoring. Soares et al. approaches the problem of preserving behavior during refactoring by analyzing a transformation and then generate a test suite for it, using static analysis. These tests are then run for both the before- and after-code, and is compared to assure that they are consistent.

1.4.3 The compositional paradigm of refactoring

This paradigm builds upon the observation of Vakilian et al. [Vak+12], that of the many automated refactorings existing in modern IDEs, the simplest ones are dominating the usage statistics. The report mainly focuses on Eclipse as the tool under investigation.

The paradigm is described almost as the opposite of automated composition of refactorings (see section 1.2.9 on page 22). It works by providing the programmer with easily accessible primitive refactorings. These refactorings shall be accessed via keyboard shortcuts or quick-assist menus[1] and be promptly executed, opposed to in the currently dominating wizard-based refactoring paradigm. They are meant to stimulate composing smaller refactorings into more complex changes, rather than doing a large upfront configuration of a wizard-based refactoring, before previewing and executing it. The compositional paradigm of refactoring is supposed to give control back to the programmer, by supporting him with an option of performing small rapid changes instead of large changes with a lesser degree of control. The report authors hope this will lead to fewer unsuccessful refactorings. It also could lower the bar for understanding the steps of a larger composite refactoring and thus also helps in figuring out what goes wrong if one should choose to opt in on a wizard-based refactoring.

Vakilian and his associates have performed a survey of the effectiveness of the compositional paradigm versus the wizard-based. They claim

[1]Think quick-assist with Ctrl+1 in Eclipse

to have found evidence of the *compositional paradigm* outperforming the *wizard-based*. It does so by reducing automation, which seems counterintuitive. Therefore they ask the question “What is an appropriate level of automation?”, and thus challenging what they feel is a rush toward more automation in the software engineering community.

Before

```java
class C {
  A a; B b;
  X x;
  void method() {
    x.y.foo();
    x.y.bar();
  }
  /* Uses of A and B.
     No uses of other
     classes. */
}

class X {
  Y y;
  /* No uses of C.
     Uses of Y. */
}

class Y {
  void foo(){
    /* No uses of C. */
  }
  void bar(){
    /* No uses of C. */
  }
}
```

After

```java
class C {
  A a; B b;
  X x;
  void method() {
    x.fooBar();
  }
  /* Uses of A and B.
     No uses of other
     classes. */
}

class X {
  Y y;
  /* No uses of C.
     Uses of Y. */
  void fooBar() {
    y.foo();
    y.bar();
  }
}

class Y {
  void foo(){
    /* No uses of C. */
  }
  void bar(){
    /* No uses of C. */
  }
}
```

Listing 7: An example of improving CBO. Class `C` has a CBO value of 4 before refactoring it, and 3 after.

Chapter 2

The search-based Extract and Move Method refactoring

In this chapter I will delve into the workings of the search-based *Extract and Move Method* refactoring. We will see the choices it must make and why it chooses a text selection as a candidate for refactoring or not.

After defining some concepts, I will introduce an example that will be used throughout the chapter to illustrate how the refactoring works in some simple situations.

2.1 The inputs to the refactoring

For executing an *Extract and Move Method* refactoring, there are two simple requirements. The first thing the refactoring needs is a text selection, telling it what to extract. Its second requirement is a target for the subsequent move operation.

When the refactoring performs changes to source code, the extracted method must be called in place of the selection that now makes up the method's body. This method call has to be performed via a variable, since the method is not static (see section 2.3.1 on the next page). Therefore, the move target must be a local variable or a field, in the scope of the text selection. The actual new location for the extracted method will be the class representing the type of the move target variable.

2.2 Defining a text selection

A text selection, in our context, is very similar to what you think of when selecting a bit of text in your editor or other text processing tool with a mouse or keyboard. It is an abstract construct that is meant to capture which specific portion of text we are about to process.

To be able to clearly reason about a text selection done to a portion of text in a computer file, which consists of pure text, we put up the following definition:

Definition. A *text selection* in a text file is defined by two non-negative integers, in addition to a reference to the file itself. The first integer is an offset into the file, while the second integer is the length of the text selection.

This means that the selected text consists of a number of characters equal to the length of the selection, where the first character is found at the specified offset.

2.3 Where we look for text selections

Next, we will see in which parts of a Java program the text selections that are analyzed and considered as candidates for the *Extract and Move Method* refactoring are found. We will also see how they are generated out of sequences of program statements.

2.3.1 Text selections are found in methods

The text selections we are interested in are those that surround program statements. Therefore, the place we look for selections that can form candidates for an execution of the *Extract and Move Method* refactoring, is within the body of a single method.

On ignoring static methods. In this project we are not analyzing static methods for candidates to the *Extract and Move Method* refactoring. The reason for this is that in the cases where we want to perform the refactoring for a selection within a static method, the first step is to extract the selection into a new method. Hence this method also becomes static, since it must be possible to call it from a static context. It would then be difficult to move the method to another class, make it non-static and calling it through a variable. To avoid these obstacles, we simply ignore static methods.

2.3.2 The possible text selections of a method body

The number of possible text selections that can be generated from the text in a method body, are equal to all the sub-sequences of characters within it. For our purposes, analyzing program source code, we must define what it means for a text selection to be valid.

Definition. A *valid text selection* is a text selection that contains all of one or more consecutive program statements.

For a sequence of statements, the text selections that can be generated from it, are equal to all its sub-sequences. Listing 9 on page 40 shows an example of all the text selections that can be generated from the code in listing 8, lines 16–18. For convenience, and the clarity of this example, the text selections are represented as tuples with the start and end line of all selections: $\{(16), (17), (18), (16, 17), (16, 18), (17, 18)\}$.

Clean

```
1  class C {
2    A a; B b; boolean bool;
3
4    void method(int val) {
5      if (bool) {
6        a.foo();
7        a = new A();
8        a.bar();
9      }
10
11     a.foo();
12     a.bar();
13
14     switch (val) {
15     case 1:
16       b.a.foo();
17       b.a.bar();
18       break;
19     default:
20       a.foo();
21     }
22   }
23 }
```

With statement sequences

```
1  class C {
2    A a; B b; boolean bool;
3
4    void method(int val) {
5      if (bool) {
6        a.foo();
7        a = new A();
8        a.bar();
9      }
10
11     a.foo();
12     a.bar();
13
14     switch (val) {
15     case 1:
16       b.a.foo();
17       b.a.bar();
18       break;
19     default:
20       a.foo();
21     }
22   }
23 }
```

Listing 8: Classes **A** and **B** are both public. The methods **foo** and **bar** are public members of class **A**.

Each nesting level of a method body can have many such sequences of statements. The outermost nesting level has one such sequence, and each branch contains its own sequence of statements. Listing 8 has a version of some code where all such sequences of statements are highlighted for a method body.

To complete our example of possible text selections, I will now list all possible text selections for the method in listing 8, by nesting level. There are 23 of them in total.

Level 1 (10 selections)

$\{(5,9),(11),(12),(14,21),(5,11),(5,12),(5,21),(11,12),(11,21),$
$(12,21)\}$

Level 2 (13 selections)

$\{(6),(7),(8),(6,7),(6,8),(7,8),(16),(17),(18),(16,17),(16,18),$
$(17,18),(20)\}$

```
        b.a.foo();
        b.a.bar();
        break;
```

Listing 9: Example of how text selections are generated for a sequence of statements. Each highlighted rectangle represents a text selection.

How many text selections are analyzed?

The complexity of how many text selections that needs to be analyzed for a body of in total n statements, is bounded by $O(n^2)$. A body of statements is here all the statements in all nesting levels of a sequence of statements. A method body (or a block) is a body of statements. To prove that the complexity is bounded by $O(n^2)$, I present a couple of theorems and prove them.

Theorem. The number of text selections that needs to be analyzed for each sequence of statements of length n, is exactly

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Proof. For $n = 1$ this is trivial: $\frac{1(1+1)}{2} = \frac{2}{2} = 1$. One statement equals one selection.

For $n = 2$, you get one text selection for the first statement, one selection for the second statement, and one selection for the two of them combined. This equals three selections. $\frac{2(2+1)}{2} = \frac{6}{2} = 3$.

For $n = 3$, you get 3 selections for the two first statements, as in the case where $n = 2$. In addition you get one selection for the third statement itself, and two more statements for the combinations of it with the two previous statements. This equals six selections. $\frac{3(3+1)}{2} = \frac{12}{2} = 6$.

Assume that for $n = k$ there exists $\frac{k(k+1)}{2}$ text selections. Then we want to add selections for another statement, following the previous k statements. So, for $n = k + 1$, we get one additional selection for the statement itself. Then we get one selection for each pair of the new selection and the previous k statements. So the total number of selections will be the number of already generated selections, plus k for every pair, plus one for the statement itself: $\frac{k(k+1)}{2} + k + 1 = \frac{k(k+1)+2k+2}{2} = \frac{k(k+1)+2(k+1)}{2} = \frac{(k+1)(k+2)}{2} = \frac{(k+1)((k+1)+1)}{2} = \sum_{i=1}^{k+1} i$ $\square$

Theorem. The number of text selections for a body of statements is maximized if all the statements are at the same level.

Proof. Assume we have a body of, in total, k statements. Then, the sum of the lengths of all the sequences of statements in the body, is also k. Let $\{l, \ldots, m, (k - l - \ldots - m)\}$ be the lengths of the sequences of statements in the body, with $l + \ldots + m < k \Rightarrow \forall i \in \{l, \ldots, m\} : i < k$.

Then, the number of text selections that are generated for the k statements is

$$\frac{l(l+1)}{2} + \ldots + \frac{m(m+1)}{2} + \frac{(k-l-\ldots-m)((k-l-\ldots-m)+1)}{2} =$$
$$\frac{l^2+l}{2} + \ldots + \frac{m^2+m}{2} + \frac{k^2-2kl-\ldots-2km+l^2+\ldots+m^2+k-l-\ldots-m}{2} =$$
$$\frac{2l^2-2kl+\ldots+2m^2-2km+k^2+k}{2}$$

It then remains to show that this inequality holds:

$$\frac{2l^2-2kl+\ldots+2m^2-2km+k^2+k}{2} < \frac{k(k+1)}{2} = \frac{k^2+k}{2}$$

By multiplication by 2 on both sides, and by removing the equal parts, we get

$$2l^2-2kl+\ldots+2m^2-2km < 0$$

Since $\forall i \in \{l,\ldots,m\} : i < k$, we have that $\forall i \in \{l,\ldots,m\} : 2ki > 2i^2$, so all the pairs of parts on the form $2i^2 - 2ki$ are negative. In sum, the inequality holds.

□

Therefore, the complexity for the number of selections that needs to be analyzed for a body of n statements is $O\left(\frac{n(n+1)}{2}\right) = O(n^2)$.

2.4 Disqualifying a selection

Certain text selections would lead to broken code if used as input to the *Extract and Move Method* refactoring. To avoid this, we have to check all text selections for such conditions before they are further analyzed. This section is therefore going to present some properties that make a selection unsuitable for our refactoring. When analyzing all these properties, it is assumed that the source code does not contain any compilation errors.

2.4.1 A call to a protected or package-private method

If a text selection contains a call to a protected or package-private method, it would not be safe to move it to another class. The reason for this, is that we cannot know if the called method is being overridden by some subclass of the enclosing class, or not.

Imagine that the protected method `foo` is declared in class A, and overridden in class B. The method `foo` is called from within a selection done to a method in A. We want to extract and move this selection to another class. The method `foo` is not public, so the *Move Method* refactoring must make it public, making the extracted method able to call it from the extracted method's new location. The problem is, that the now public method `foo` is overridden in a subclass, where it has a protected status. This makes the compiler complain that the subclass B is trying to reduce the visibility of a method declared in its superclass A. This is not allowed in Java, and for good reasons. It would make it possible to create a subclass that could not be a substitute for its superclass.

The problem this check helps to avoid, is a little subtle. The problem does not arise in the class where the change is done, but in a class derived from it. This shows that classes acting as superclasses are especially fragile to errors introduced by automated refactorings.

2.4.2 A double class instance creation

The following is a problem caused solely by the underlying *Move Method* refactoring. The problem occurs if two classes are instantiated such that the first constructor invocation is an argument to a second, and that the first constructor invocation takes an argument that is built up using a field. As an example, say that `name` is a field of the enclosing class, and we have the expression `new A(new B(name))`. If this expression is located in a selection that is moved to another class, `name` will be left untouched, instead of being prefixed with a variable of the same type as it is declared in. If `name` is the destination for the move, it is not replaced by `this`, or removed if it is a prefix to a member access (`name.member`), but it is still left by itself.

Situations like this would lead to code that will not compile. Therefore, we have to avoid them by not allowing selections to contain such double class instance creations that also contain references to fields.

2.4.3 Instantiation of non-static inner class

When a non-static inner class is instantiated, this must happen in the scope of its declaring class. This is because it must have access to the members of the declaring class. If the inner class is public, it is possible to instantiate it through an instance of its declaring class, but this is not handled by the underlying *Move Method* refactoring.

Performing a move on a method that instantiates a non-static inner class, will break the code if the instantiation is not handled properly. For this reason, selections that contain instantiations of non-static inner classes are deemed unsuitable for the *Extract and Move Method* refactoring.

2.4.4 References to enclosing instances of the enclosing class

To "reference an enclosing instance of the enclosing class" is to reference another instance than the one for the immediately enclosing class. Imagine there is a (non-static) class C that is declared in the inner scope of another class. That class can again be nested inside a third class, and so on. Hence, the nested class C can have access to many enclosing instances of its innermost enclosing class. A selection in a method declared in class C

is disqualified if it contains a statement that contains a reference to one or more instances of these enclosing classes of C.

The problem with such a reference, is that it may not be valid if it is moved to another class. Theoretically, some situations could easily be solved by passing, to the moved method, a reference to the instance where the problematic referenced member is declared. This should work in the case where this member is publicly accessible. This is not done in the underlying *Move Method* refactoring, so it cannot be allowed in the *Extract and Move Method* refactoring either.

2.4.5 Inconsistent return statements

To verify that a text selection is consistent with respect to return statements, we must check that if a selection contains a return statement, then every possible execution path within the selection ends in either a return or a throw statement. This property is important regarding the *Extract Method* refactoring. If it holds, it means that a method could be extracted from the selection, and a call to it could be substituted for the selection. If the method has a non-void return type, then a call to it would also be a valid return point for the calling method. If its return value is of the void type, then the *Extract Method* refactoring will append an empty return statement to the back of the method call. Therefore, the analysis does not discriminate on either kind of return statements, with or without a return value.

A *throw* statement is accepted anywhere a return statement is required. This is because a throw statement causes an immediate exit from the current block, together with all outer blocks in its control flow that do not catch the thrown exception.

We separate between explicit and implicit return statements. An *explicit* return statement is formed by using the **`return`** keyword, while an *implicit* return statement is a statement that is not formed using **`return`**, but must be the last statement of a method that can have any side effects. This can happen in methods with a void return type. An example is a statement that is located inside one or more blocks. The last statement of a method could for instance be a synchronized statement, but the last statement that is executed in the method, and that can have any side effects, may be located inside the body of the synchronized statement.

We can start the check for the "inconsistent return statements" property by looking at the last statement of a selection to see if it is a return statement (explicit or implicit) or a throw statement. If this is the case, then the property holds. All execution paths within the selection should end in either this, or another, return or throw statement.

If the last statement of the selection is not a *return* or *throw*, the execution of it must eventually end in one of these types of statements for the selection to be legal. This means that all branches of the last statement of every branch must end in a return or throw. Given this recursive definition, there are only five types of statements that are guaranteed to end in a return or throw if their child branches do. All other statements would have to be considered illegal. The first three: Block-statements, labeled statements and do-statements are all kinds of fall-through statements that always get their body executed. Do-statements would not make much sense if written such that they always end after the first round of execution of their body, but that is not our concern. The remaining two statements that can end in a return or throw are if-statements and try-statements.

For an if-statement, the rule is that if its then-part does not contain any return or throw statements, this is considered illegal. If the then-part does contain a return or throw, the else-part is checked. If its else-part is non-existent, or it does not contain any return or throw statements, the statement is considered illegal. If an if-statement is not considered illegal, the bodies of its two parts must be checked.

Try-statements are handled similar to if-statements. The body of a try-statement must contain a return or throw, and if it does not, its finally body must. The catch clauses of a try-statement must always end in a return or throw.

2.4.6 Ambiguous return values

The problem with ambiguous return values arises when a selection is chosen to be extracted into a new method, but if refactored it needs to return more than one value from that method.

This problem occurs in two situations. The first situation arises when there is more than one local variable that is both assigned to within a selection and also referenced after the selection. The other situation occurs when there is only one such assignment, but the selection also contains return statements.

Therefore we must examine the selection for assignments to local variables that are referenced after the text selection. Then we must verify that not more than one such reference is done, or zero if any return statements are found.

2.4.7 Illegal statements

An illegal statement may be a statement that is of a type that is never allowed, or it may be a statement of a type that is only allowed if certain conditions are true.

Any use of the `super` keyword is prohibited, since its meaning is altered when moving a method to another class.

For a *break* statement, there are two situations to consider: A break statement with or without a label. If the break statement has a label, it is checked that whole of the labeled statement is inside the selection. If the break statement does not have a label attached to it, it is checked that its innermost enclosing loop or switch statement also is inside the selection.

The situation for a *continue* statement is the same as for a break statement, except that it is not allowed inside switch statements.

Regarding *assignments*, two types of assignments are allowed: Assignments to non-final variables and assignments to array access. All other assignments are regarded illegal.

Incompleteness. The list of illegal statements is not complete, and a lot of situations that can lead to compilation errors or behavior changes are not considered. It is not feasible to consider all such situations within the limits of this master's project, and maybe not outside of it either. The feasibility of this problem could be further explored by others.

2.5 Disqualifying selections from the example

Among the selections we found for the code in listing 8 on page 39, not many of them must be disqualified on the basis of containing something illegal. The only statement causing trouble is the break statement in line 18. None of the selections on nesting level 2 can contain this break statement, since its innermost switch statement is not inside any of these selections.

This means that the text selections (18), (16, 18) and (17, 18) can be excluded from further consideration, and we are left with the following selections:

Level 1 (10 selections)

$\{(5,9),(11),(12),(14,21),(5,11),(5,12),(5,21),(11,12),(11,21),(12,21)\}$

Level 2 (10 selections)

$\{(6),(7),(8),(6,7),(6,8),(7,8),(16),(17),(16,17),(20)\}$

2.6 Finding a move target

In the analysis needed to perform the *Extract and Move Method* refactoring automatically, the selection we choose is found among all the selections that have a possible move target. Therefore, the best possible move target must be found for all the candidate selections, so that we are able to sort out the selection that is best suited for the refactoring.

To find the best move target for a specific text selection, we first need to find all the possible targets. Since the target must be a local variable or a field, we are basically looking for names within the selection; names that represents references to variables.

The names we are looking for, we call them *prefixes*. This is because we are not interested in names that occur in the middle of a dot-separated sequence of names. We are only interested in names constituting prefixes of other names, and possibly themselves. The reason for this, is that two lexically equal names need not be referencing the same variable, if they themselves are not referenced via the same prefix. Consider the two method calls `a.x.foo()` and `b.x.foo()`. Here, the two references to `x`, in the middle of the qualified names both preceding `foo()`, are not referencing the same variable. Even though the variables may share the type, and the method `foo` thus is the same for both, we would not know through which of the variables `a` or `b` we should call the extracted method.

The possible move targets are then the prefixes that are not among a subset of the prefixes that are not valid move targets (see section 2.7).

Also, prefixes that are just simple names, and have only one occurrence, are left out. This is because they are not going to have any positive effect on coupling between classes, and are only going to increase the complexity of the code.

For finding the best move target among these safe prefixes, a simple heuristic is used. It is as simple as choosing the prefix that is most frequently referenced within the selection. If two prefixes have equally many occurrences, the one with the largest number of segments is preferred. This is because we want to favor indirection, as it may lower coupling between classes.

2.7 Unfixes

We will call the prefixes that are not valid as move targets for *unfixes*.

Assignments. A name that is assigned to within a selection is an unfix. The reason for this is that the result would be an assignment to the **`this`** keyword, which is not valid in Java (see appendix A.1 on page 109).

Variable declarations. Prefixes that originate from variable declarations within the same selection are also considered unfixes. The reason for this is that when a method is moved, it needs to be called through a variable. If this variable is also declared within the method that is to be moved, this obviously cannot be done.

Unmodifiable types. Also considered as unfixes are variable references that are of types that are not suitable for moving methods to. This can either be because it is not physically possible to move a method to a class or that it will cause compilation errors by doing so.

If the type binding for a name is not resolved it is considered an unfix. The same applies to types that are only found in compiled code, so they have no underlying source that is accessible to us. (E.g. the **`java.lang.String`** class.)

Nor are interface types suitable as targets. This is simply because interfaces in Java cannot contain methods with bodies. (This thesis does not deal with features of Java versions later than Java 7. Java 8 has interfaces with default implementations of methods.)

Local types. Neither are local types allowed. This accounts for both local and anonymous classes. Anonymous classes are effectively the same as interface types with respect to unfixes. Local classes could in theory be used as targets, but this is not possible due to limitations of the way the *Extract and Move Method* refactoring has to be implemented. The problem is that the refactoring is done in two steps, so the intermediate state between the two refactorings would not be legal Java code. In the intermediate step for the case where a local class is the move target, the extracted method would need to take the local class as a parameter. This new method would need to live in the scope of the declaring class of the originating method. The local class would then not be in the scope of the extracted method, thus bringing the source code into an illegal state. This scenario is shown in listing 10. One could imagine that the method was extracted and moved in one operation, without an intermediate state. Then it would make sense to include variables with types of local classes in the set of legal targets, since the local classes would then be in the scopes of the method calls. If this makes any difference for software metrics that measure coupling would be a different discussion.

Before

```
void declaresLocalClass() {
  class LocalClass {
    void foo() {}
    void bar() {}
  }

  LocalClass inst =
    new LocalClass();
  inst.foo();
  inst.bar();
}
```

After Extract Method

```
void declaresLocalClass() {
  class LocalClass {
    void foo() {}
    void bar() {}
  }

  LocalClass inst =
    new LocalClass();
  fooBar(inst);
}

// Illegal intermediate step
void fooBar(LocalClass inst) {
  inst.foo();
  inst.bar();
}
```

Listing 10: The *Extract and Move Method* refactoring bringing the code into an illegal state with an intermediate step.

The last class of names that are considered unfixes are names used in null tests. These are tests that read like this: if **`<name>`** equals **`null`** then do something. If allowing variables used in those kinds of expressions as targets for moving methods, we would end up with code containing boolean expressions like **`this == null`**, which would always evaluate to **`false`**, since **`this`** would never be **`null`**. The existence of a null test indicates that a variable is expected to sometimes hold the value **`null`**. By choosing a variable used in a null test as a move target, we could potentially end up with a null pointer exception if the extracted and moved method is called on a variable with a null value.

2.8 Finding the example selections that have possible targets

We now pick up the thread from section 2.5 on page 45 where we have a set of text selections that needs to be analyzed to find out if some of them are suitable targets for the *Extract and Move Method* refactoring.

We start by analyzing the text selections for nesting level 2, because these results can be used to reason about the selections for nesting level 1. First we have all the single-statement selections:

Selections (6)**,** (8) **and** (20)**.**
: All these selections have a prefix that contains a possible target, namely the variable `a`. The problem is that the prefixes are only one segment long, and their frequency counts are only 1 as well. None of these selections are therefore considered as suitable candidates for the refactoring.

Selection (7)**.**
: This selection contains the unfix `a`, and no other possible targets. The reason for `a` being an unfix is that it is assigned to within the selection. Selection (7) is therefore unsuited as a refactoring candidate.

Selections (16) **and** (17)**.**
: These selections have possible targets. The target for both selections is the variable `b`. Both the prefixes have frequency 1. We denote this with the new tuples $((16), \texttt{b.a}, f(1))$ and $((17), \texttt{b.a}, f(1))$. They contain the selection, the prefix with the target and the frequency for this prefix.

Then we have all the text selections from level 2 that are composed of multiple statements:

Selections $(6,7)$**,** $(6,8)$ **and** $(7,8)$**.**
: All these selections are disqualified for the reason that they contain the unfix `a`, due to the assignment, and no other possible move targets.

Selection $(16,17)$**.**
: This is the last selection we analyze on nesting level 2. It contains only one possible move target, which is the variable `b`. It also contains only one prefix `b.a`, with frequency count 2. Therefore, we have a new candidate $((16,17), \texttt{b.a}, f(2))$.

We now move on to the text selections for nesting level 1, starting with the single-statement selections:

Selection $(5,9)$**.**
: This selection contains two variable references that must be analyzed to see if they are possible move candidates. The first one is the variable `bool`. This variable is of type `boolean`, which is a primary type and

therefore not possible to make any changes to. The second variable is `a`. The variable `a` is an unfix in $(5, 9)$, for the same reason as in the selections $(6, 7)$, $(7, 8)$ and $(6, 8)$. So selection $(5, 9)$ contains no possible move targets.

Selections (11) and (12).
These selections are disqualified for the same reasons as selections (6) and (8). Their prefixes are one segment long and are referenced only one time.

Selection $(14, 21)$
This is the switch statement from listing 8 on page 39. It contains the relevant variable references `val`, `a` and `b`. The variable `val` is a primary type, just as `bool`. The variable `a` is only found in one statement, and in a prefix with only one segment, so it is not considered to be a possible move target. The only variable left is `b`. Just as in the selection $(16, 17)$, `b` is part of the prefix `b.a`, which has 2 appearances. We have found a new candidate $((14, 21), \texttt{b.a}, f(2))$.

It remains to see if we find any new candidates by analyzing the multi-statement text selections for nesting level 1:

Selections $(5, 11)$ and $(5, 12)$.
These selections are disqualified for the same reason as $(5, 9)$. The only possible move target `a` is an unfix.

Selection $(5, 21)$.
This is whole of the method body in listing 8. The variables `a`, `bool` and `val` are either unfixes or primary types. The variable `b` is the only possible move target, and again we have the prefix `b.a` as the center of attention. The refactoring candidate is $((5, 21), \texttt{b.a}, f(2))$.

Selection $(11, 12)$.
This small selection contains the prefix `a` with frequency 2, and no unfixes. The candidate is $((11, 12), \texttt{a}, f(2))$.

Selection $(11, 21)$
This selection contains the selection $(11, 12)$ in addition to the switch statement. The selection has two possible move targets. The first one is `b`, in a prefix with frequency 2. The second is `a`, although it is in a simple prefix, it is referenced 3 times, and is therefore chosen as the target for the selection. The new candidate is $((11, 21), \texttt{a}, f(3))$.

Selection $(12, 21)$.
This selection is similar to the previous $(11, 21)$, only that `a` now has frequency count 2. This means that the prefixes `a` and `b.a` both have the count 2. Of the two, `b.a` is preferred, since it has more segments than `a`. Thus the candidate for this selection is $((12, 21), \texttt{b.a}, f(2))$.

For the method in listing 8 on page 39 we therefore have the following 8 candidates for the *Extract and Move Method* refactoring:

$\{((16), \texttt{b.a}, f(1)), ((17), \texttt{b.a}, f(1)), ((16, 17), \texttt{b.a}, f(2)), ((14, 21), \texttt{b.a}, f(2)),$
$((5, 21), \texttt{b.a}, f(2)), ((11, 12), \texttt{a}, f(2)), ((11, 21), \texttt{a}, f(3)), ((12, 21), \texttt{b.a}, f(2))\}$.

It now only remains to specify an order for these candidates, so we can choose the most suitable one to refactor.

2.9 Choosing between selections

When choosing a selection between the text selections that have possible move targets, the selections need to be ordered. The criteria below are presented in the order they are prioritized. If not one selection is favored over the other for a concrete criterion, the selections are evaluated by the next criterion.

1. The first criterion that is evaluated is whether a selection contains any unfixes or not. If selection A contains no unfixes, while selection B does, selection A is favored over selection B. This is because, if we can, we want to avoid moving such as assignments and variable declarations. This is done under the assumption that avoiding selections containing unfixes, if possible, will make the moved code a little cleaner.

2. The second criterion that is evaluated is whether a selection contains multiple possible targets or not. If selection A has only one possible target, and selection B has multiple, selection A is favored. If both selections have multiple possible targets, they are considered equal with respect to this criterion. The rationale for this heuristic is that we would prefer not to introduce new couplings between classes when performing the *Extract and Move Method* refactoring.

3. When evaluating this next criterion, it is with the knowledge that selection A and B both have a possible target. Then, if the move target candidate of selection A has a higher reference count than the target candidate of selection B, selection A is favored. The reason for this is that we would like to move the selection that gets rid of the most references to another class.

4. The last criterion is that if the frequencies of the targets chosen for both selections are equal, the selection with the target that is part of the prefix with highest number of segments is favored. This is done to favor indirection.

If none of the above mentioned criteria favor one selection over another, the selections are considered to be equally good candidates for the *Extract and Move Method* refactoring.

2.10 Performing changes

When a text selection and a move target is found for the *Extract and Move Method* refactoring, the actual changes are executed by two existing primitive refactorings. First the *Extract Method* refactoring is used to extract the selection into a new method. Then the *Move Method* refactoring is used to move that new method to the class determined by the move target.

If, at any point, an exception is thrown or the preconditions for one of the primitive refactorings are not satisfied, the composite refactoring is aborted, and the source code is left in its current state. This has the implication that the *Extract and Move Method* refactoring could end up being partially executed. This happens if the *Extract Method* refactoring is executed, but the *Move Method* refactoring is being canceled. A partial execution is not considered a problem, since the code should still compile.

2.11 Concluding the example

For choosing one of the remaining selections, we need to order our candidates after the criteria in the previous section. Below we have the candidates ordered in descending order, with the "best" candidate first:

1. $((16,17), \texttt{b.a}, f(2))$
2. $((11,12), \texttt{a}, f(2))$
3. $((16), \texttt{b.a}, f(1))$
4. $((17), \texttt{b.a}, f(1))$
5. $((11,21), \texttt{a}, f(3))$
6. $((5,21), \texttt{b.a}, f(2))$
7. $((12,21), \texttt{b.a}, f(2))$
8. $((14,21), \texttt{b.a}, f(2))$

The candidates ordered 5–8 all have unfixes in them, therefore they are ordered last. None of the candidates ordered 1–4 have multiple possible targets. The only interesting discussion is now why $(16,17)$ was favored over $(11,12)$. This is because the prefix `b.a` enclosing the move target of selection $(16,17)$ has one more segment than the prefix `a` of $(11,12)$.

The selection $(16,17)$ is now extracted into a new method `gen_123` and then moved to class `B`, since that is the type of the variable `b` that is the move target from the chosen refactoring candidate. The name of the method has a randomly generated suffix. In the refactoring implementation, the extracted methods follow the pattern `generated_<long>`, where `<long>` is a pseudo-random long value. This is shortened here to make the example readable. The result after the refactoring is shown in listing 11 on the following page.

Implementation details. Implementation details for the various parts of this chapter are found in chapter 4 on page 65.

```
1  class C {
2    A a; B b; boolean bool;
3
4    void method(int val) {
5      if (bool) {
6        a.foo();
7        a = new A();
8        a.bar();
9      }
10
11     a.foo();
12     a.bar();
13
14     switch (val) {
15     case 1:
16       b.gen_123(this);
17       break;
18     default:
19       a.foo();
20     }
21   }
22 }
```

```
public class B {
  A a;

  public void gen_123(C c) {
    a.foo();
    a.bar();
  }
}
```

Listing 11: The result after refactoring the class `C` of listing 8 on page 39 with the *Extract and Move Method* refactoring, with $((16, 17), \texttt{b.a}, f(2))$ as input.

Chapter 3

The Eclipse Platform with the Java development tools

The Eclipse Platform is an extensible platform. It can be used to build IDEs for many programming languages. For it to be a fully functional Java IDE, it must be equipped with the Java development tools plugin, abbreviated as JDT.

This chapter will present how to analyze and change Java source code by utilizing the APIs supplied by Eclipse and the JDT.

3.1 Analyzing source code in Eclipse

In this section we will see how to access Java source code in the Eclipse workspace. Then it is shown how this code is being represented when it is parsed and how to search this representation for the properties we are after.

3.1.1 The Java model

The Java model of Eclipse is its internal representation of a Java project. It is light-weight, and has only limited possibilities for manipulating source code. It is typically used as a basis for the Package Explorer in Eclipse.

The elements of the Java model are only handles to the underlying elements. This means that the underlying element of a handle does not need to actually exist. Hence, the user of a handle must always check that it exist by calling the `exists` method of the handle.

The different handles with descriptions are listed in table 3.1 on the following page, while the hierarchy of the Java Model is shown in figure 3.1 on page 55.

3.1.2 The abstract syntax tree

Eclipse is following the common paradigm of using an abstract syntax tree for source code analysis and manipulation.

When parsing program source code into something that can be used as a foundation for analysis, the start of the process follows the same steps as in a compiler. This is all natural, because the way a compiler analyzes code is

Table 3.1: The elements of the Java Model [Vog12].

Project Element	**Java Model element**	**Description**
Java project	`IJavaProject`	The Java project which contains all other objects.
Source folder / binary folder / external library	`IPackageFragmentRoot`	Hold source or binary files, can be a folder or a library (zip / jar file).
Each package	`IPackageFragment`	Each package is below the `IPackageFragmentRoot`, sub-packages are not leaves of the package, they are listed directed under `IPackageFragmentRoot`.
Java Source file	`ICompilationUnit`	The Source file is always below the package node.
Types / Fields / Methods	`IType` / `IField` / `IMethod`	Types, fields and methods.

no different from how source manipulation programs would do it, except for some properties of code that are analyzed in the parser. Thus, the process of translating source code into a structure that is suitable for analyzing, can be seen as a kind of interrupted compilation process (see figure 3.2 on page 56).

The process starts with a *scanner*, or lexer. The job of the scanner is to read the source code and divide it into tokens for the parser. Therefore, it is also sometimes called a tokenizer. A token is a logical unit, defined in the language specification, consisting of one or more consecutive characters. In the Java language the tokens can for instance be the **`this`** keyword, a curly bracket **`{`** or a **`nameToken`**. It is recognized by the scanner on the basis of something equivalent of a regular expression. This part of the process is often implemented with the use of a finite automaton. In fact, it is common to specify the tokens in regular expressions, which in turn are translated into a finite automaton lexer. This process can be automated.

The program component used to translate a stream of tokens into something meaningful, is called a parser. A parser is fed tokens from the scanner and performs an analysis of the structure of a program. It verifies that the syntax is correct according to the grammar rules of a language, which are usually specified in a context-free grammar, and often in a variant of the Backus–Naur Form. The result coming from the parser is in the form of an *Abstract Syntax Tree*, AST for short. It is called *abstract*, because the structure does not contain all of the tokens produced by the scanner.

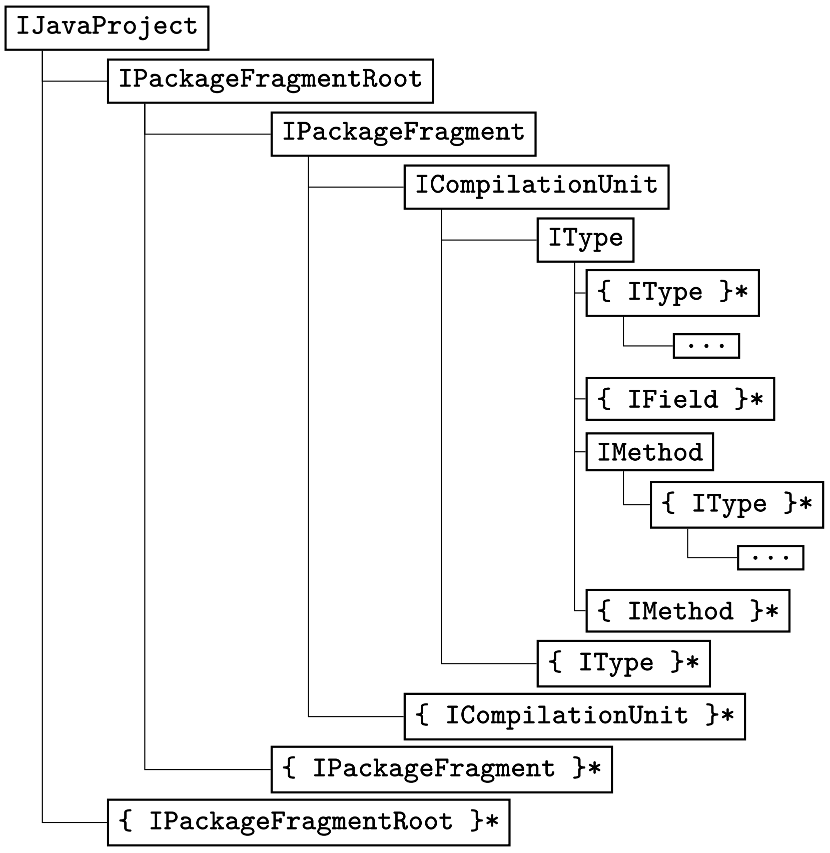

Figure 3.1: The Java model of Eclipse. "`{ SomeElement }*`" means "`SomeElement` zero or more times". For recursive structures, "`...`" is used.

It only contains logical constructs, and because it forms a tree, all kinds of parentheses and brackets are implicit in the structure. It is this AST that is used when performing the semantic analysis of the code.

As an example, we can think of the expression `(5 + 7) * 2`. The root of this tree would in Eclipse be an `InfixExpression` with the operator `TIMES`, and a left operand, which is also an `InfixExpression` with the operator `PLUS`. The left operand `InfixExpression`, has in turn a left operand of type `NumberLiteral` with the value `"5"` and a right operand `NumberLiteral` with the value `"7"`. The root will have a right operand of type `NumberLiteral` and value `"2"`. The AST for this expression is illustrated in figure 3.3 on the next page.

Contrary to the Java Model, an abstract syntax tree is a heavy-weight representation of source code. It contains information about properties like type bindings for variables and variable bindings for names.

The AST in Eclipse

In Eclipse, every node in the AST is a child of the abstract superclass `ASTNode`[1]. Every `ASTNode`, among a lot of other things, provides information

[1]`org.eclipse.jdt.core.dom.ASTNode`

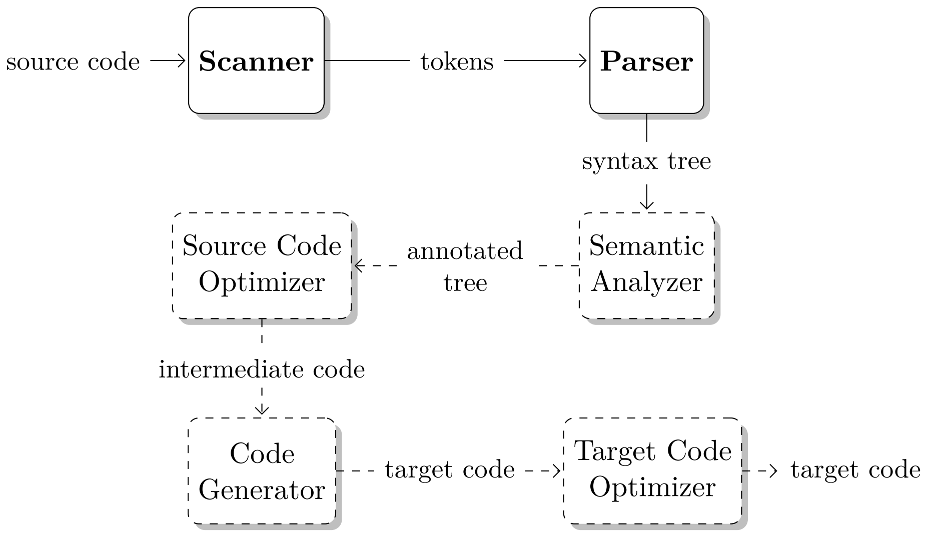

Figure 3.2: Interrupted compilation process. (Full compilation process borrowed from *Compiler construction: principles and practice* by Kenneth C. Louden [Lou97].)

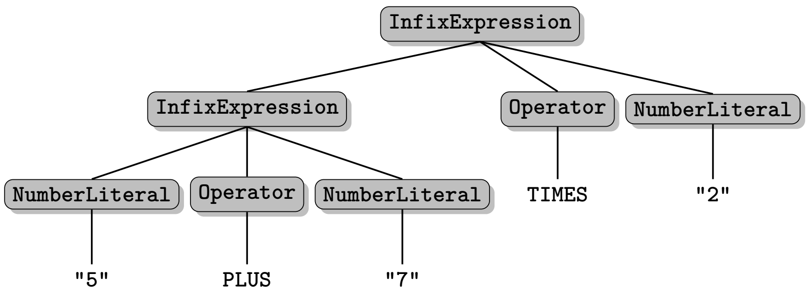

Figure 3.3: The abstract syntax tree for the expression `(5 + 7) * 2`.

about its position and length in the source code, as well as a reference to its parent and to the root of the tree.

The root of the AST is always of type **`CompilationUnit`**. It is not the same as an instance of an **`ICompilationUnit`**, which is the compilation unit handle of the Java model. The children of a **`CompilationUnit`** is an optional **`PackageDeclaration`**, zero or more nodes of type **`ImportDecaration`** and all its top-level type declarations that has node types **`AbstractTypeDeclaration`**.

An **`AbstractTypeDeclaration`** can be one of the types **`AnnotationType-Declaration`**, **`EnumDeclaration`** or **`TypeDeclaration`**. The children of an **`AbstractTypeDeclaration`** must be a subtype of **`BodyDeclaration`**. These subtypes are: **`AnnotationTypeMemberDeclaration`**, **`EnumConstant-Declaration`**, **`FieldDeclaration`**, **`Initializer`** and **`MethodDeclaration`**.

Of the body declarations, the **`MethodDeclaration`** is the most interesting one. Its children include lists of modifiers, type parameters, parameters and exceptions. It has a return type node and a body node. The body, if present, is of type **`Block`**. A **`Block`** is itself a **`Statement`**, and its children is a list of

Statement nodes.

There are too many subtypes of the abstract type **Statement** to list up, but there exists a subtype of **Statement** for every statement type of Java, as one would expect. This also applies to the abstract type **Expression**. However, the expression **Name** is a little special, since it is both used as an operand in compound expressions, as well as for names in type declarations and such.

There is an overview of some of the structure of an Eclipse AST in figure 3.4.

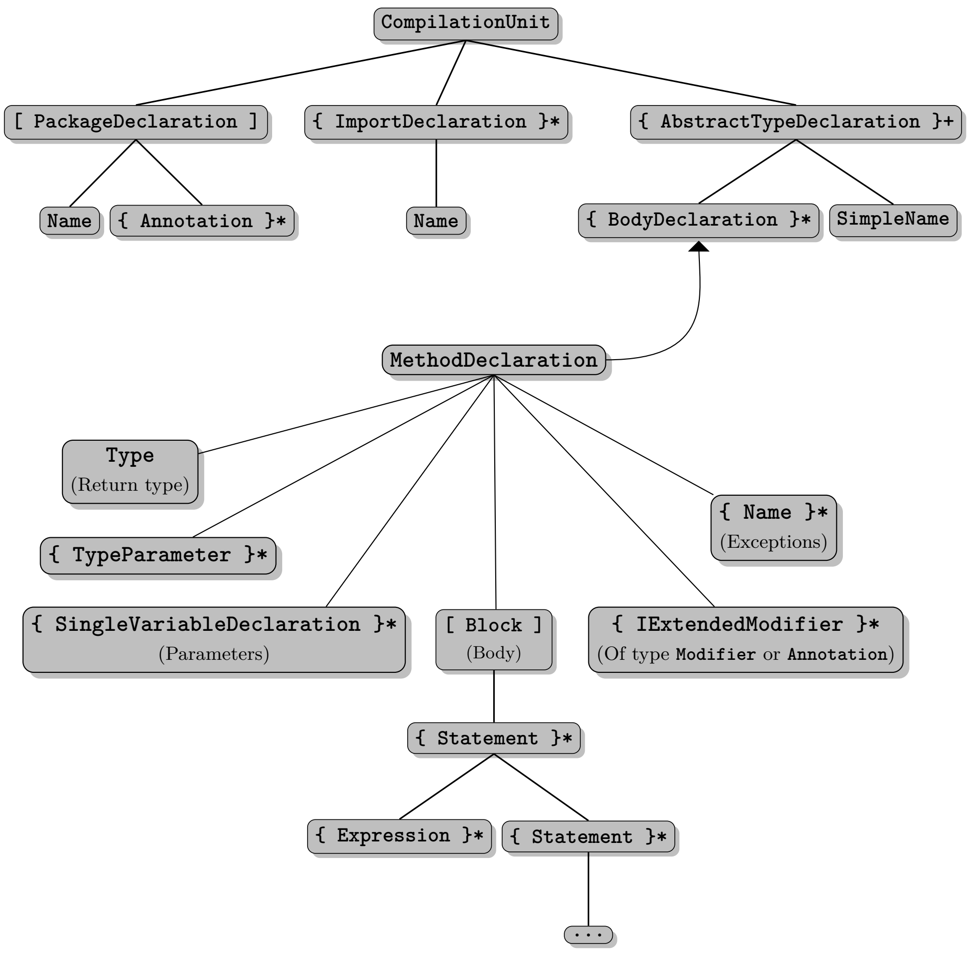

Figure 3.4: The format of the abstract syntax tree in Eclipse.

3.1.3 The ASTVisitor

So far, the only thing that has been addressed is how the data that is going to be the basis for our analysis is structured. Another aspect of it is how we are going to traverse the AST to gather the information we need. It is of course possible to start at the top of the tree, and manually search through its nodes for the ones we are looking for, but that is a bit inconvenient. To be able to efficiently utilize such an approach, we would need to make our own framework for traversing the tree and visiting only the types of nodes we are after. Luckily, this functionality is already present in Eclipse, by its **`ASTVisitor`**[1].

The Eclipse AST, together with its **`ASTVisitor`**, follows the *Visitor* pattern [Gam+95]. The intent of this design pattern is to facilitate extending the functionality of classes without touching the classes themselves.

Let us say that there is a class hierarchy of elements. These elements all have a method **`accept(Visitor visitor)`**. In its simplest form, the **`accept`** method just calls the **`visit`** method of the visitor with the node itself as an argument, like this: **`visitor.visit(this)`**. For the visitors to be able to extend the functionality of all the classes in the elements hierarchy, each **`Visitor`** must have one visit method for each concrete class in the hierarchy. Say the hierarchy consists of the concrete classes **`ConcreteElementA`** and **`ConcreteElementB`**. Then each visitor must have the (possibly empty) methods **`visit(ConcreteElementA element)`** and **`visit(ConcreteElementB element)`**. This scenario is depicted in figure 3.5.

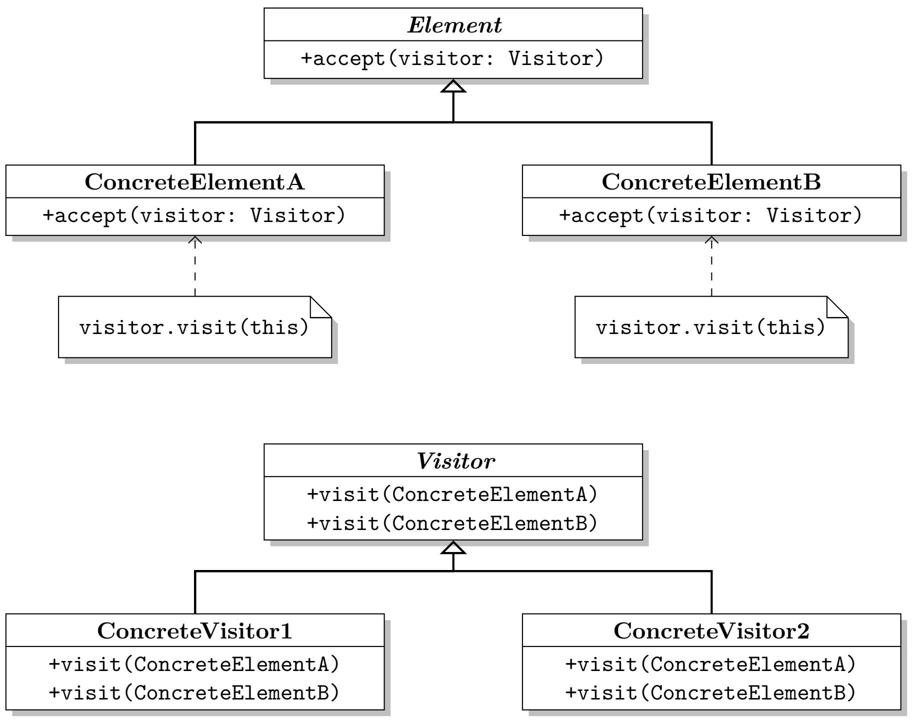

Figure 3.5: The Visitor Pattern.

The use of the visitor pattern can be appropriate when the hierarchy of elements is mostly stable, but the family of operations over its elements is constantly growing. This is clearly the case for the Eclipse AST, since the hierarchy for the type **`ASTNode`** is very stable, but the functionality of its elements is extended every time someone need to operate on the AST. Another aspect of the Eclipse implementation is that it is a public API, and

[1] `org.eclipse.jdt.core.dom.ASTVisitor`

the visitor pattern is an easy way to provide access to the nodes in the tree.

The version of the visitor pattern implemented for the AST nodes in Eclipse also provides an elegant way to traverse the tree. It does so by following the convention that every node in the tree first let the visitor visit itself, before it also makes all its children accept the visitor. The children are only visited if the visit method of their parent returns **`true`**. This pattern then makes for a prefix traversal of the AST. If postfix traversal is desired, the visitors also have **`endVisit`** methods for each node type, which is called after the **`visit`** method for a node. In addition to these visit methods, there are also the methods **`preVisit(ASTNode)`**, **`postVisit(ASTNode)`** and **`preVisit2(ASTNode)`**. The **`preVisit`** method is called before the type-specific **`visit`** method. The **`postVisit`** method is called after the type-specific **`endVisit`**. The type specific **`visit`** is only called if **`preVisit2`** returns **`true`**. Overriding the **`preVisit2`** also alters the behavior of **`preVisit`**, since the default implementation of **`preVisit2`** is responsible for calling **`preVisit`**.

An example of a trivial **`ASTVisitor`** is shown in listing 12.

```
public class CollectNamesVisitor extends ASTVisitor {
    Collection<Name> names = new LinkedList<Name>();

    @Override
    public boolean visit(QualifiedName node) {
      names.add(node);
      return false;
    }

    @Override
    public boolean visit(SimpleName node) {
        names.add(node);
        return true;
    }
}
```

Listing 12: An **`ASTVisitor`** that visits all the names in a subtree and adds them to a collection, except those names that are children of any **`QualifiedName`**.

3.2 The refactoring API of Eclipse

This section will present the design behind the refactoring support in Eclipse, and the JDT in specific. After which it will follow a section about shortcomings of the refactoring API in terms of composition of refactorings.

3.2.1 Design

The refactoring world of Eclipse can in general be separated into two parts: The language independent part and the part written for a specific programming language – the language that is the target of the supported Java refactorings.

The Language Toolkit.

The Language Toolkit[1], or LTK for short, is the framework that is used to implement refactorings in Eclipse. It is language independent and provides the abstractions of a refactoring and the change it generates, in the form of the classes **`Refactoring`**[2] and **`Change`**[3].

There are also parts of the LTK that is concerned with user interaction, but they will not be discussed here, since they are of little value to us and our use of the framework. We are primarily interested in the parts that can be automated.

The Refactoring class. The abstract class **`Refactoring`** is the core of the LTK framework. Every refactoring that is going to be supported by the LTK has to end up creating an instance of one of its subclasses. The main responsibilities of subclasses of **`Refactoring`** are to implement template methods for precondition checking (**`checkInitialConditions`**[4] and **`checkFinalConditions`**[5]), in addition to the **`createChange`**[6] method that creates and returns an instance of the **`Change`** class.

If the refactoring shall support that others participate in it when it is executed, the refactoring has to be a processor-based refactoring[7]. It then delegates to its given **`RefactoringProcessor`**[8] for condition checking and change creation. Participating in a refactoring can be useful in cases where the changes done to programming source code affect other related resources in the workspace. This can be names or paths in configuration files, or maybe one would like to perform additional logging of changes done in the workspace.

The Change class. This class is the base class for objects that are responsible for performing the actual workspace transformations in a refactoring. The main responsibilities for its subclasses are to implement

[1]The content of this section is a mixture of written material from https://www.eclipse.org/articles/Article-LTK/ltk.html and http://www.eclipse.org/articles/article.php?file=Article-Unleashing-the-Power-of-Refactoring/index.html, the LTK source code and my own memory.

[2]`org.eclipse.ltk.core.refactoring.Refactoring`

[3]`org.eclipse.ltk.core.refactoring.Change`

[4]`org.eclipse.ltk.core.refactoring.Refactoring#checkInitialConditions()`

[5]`org.eclipse.ltk.core.refactoring.Refactoring#checkFinalConditions()`

[6]`org.eclipse.ltk.core.refactoring.Refactoring#createChange()`

[7]`org.eclipse.ltk.core.refactoring.participants.ProcessorBasedRefactoring`

[8]`org.eclipse.ltk.core.refactoring.participants.RefactoringProcessor`

the `perform`[1] and `isValid`[2] methods. The `isValid` method verifies that the change object is valid and thus can be executed by calling its `perform` method. The `perform` method performs the desired change and returns an undo change object that can be used to reverse the effect of the transformation done by its originating change object.

Executing a refactoring The life-cycle of an LTK refactoring generally follows two steps after creation: condition checking and change creation. By letting the refactoring object be handled by a `CheckConditionsOperation`[3] that in turn is handled by a `CreateChangeOperation`[4], it is assured that the change creation process is managed in a proper manner.

The actual execution of a change object has to follow a detailed life cycle. This life cycle is honored if the `CreateChangeOperation` is handled by a `PerformChangeOperation`[5]. If also an undo manager[6] is set for the `PerformChangeOperation`, the undo change is added into the undo history.

The language specific refactorings

The language specific refactorings supplied by the JDT that are relevant for this project are presented below. It is the JDT-implementations of the two primitive refactorings *Extract Method* and *Move Method*. In the JDT, the implementations of these refactorings are found in the classes `ExtractMethodRefactoring`[7] and `MoveInstanceMethodProcessor`[8], where the last class is designed to be used together with the processor-based `MoveRefactoring`[9].

The ExtractMethodRefactoring class. This class is quite simple in its use. The only parameters it requires for construction is a compilation unit[10], the offset into the source code where the extraction shall start, and the length of the source to be extracted. Then you have to set the method name for the new method together with its visibility and some not so interesting parameters.

The MoveInstanceMethodProcessor class. For the *Move Method* refactoring, the processor requires a little more advanced input than the class for the *Extract Method* refactoring. For construction it requires a method handle[11] for the method that is to be moved. Then the target for

[1]`org.eclipse.ltk.core.refactoring.Change#perform()`
[2]`org.eclipse.ltk.core.refactoring.Change#isValid()`
[3]`org.eclipse.ltk.core.refactoring.CheckConditionsOperation`
[4]`org.eclipse.ltk.core.refactoring.CreateChangeOperation`
[5]`org.eclipse.ltk.core.refactoring.PerformChangeOperation`
[6]`org.eclipse.ltk.core.refactoring.IUndoManager`
[7]`org.eclipse.jdt.internal.corext.refactoring.code.ExtractMethodRefactoring`
[8]`org.eclipse.jdt.internal.corext.refactoring.structure.MoveInstanceMethod-Processor`
[9]`org.eclipse.ltk.core.refactoring.participants.MoveRefactoring`
[10]`org.eclipse.jdt.core.ICompilationUnit`
[11]`org.eclipse.jdt.core.IMethod`

the move has to be supplied as the variable binding from a chosen variable declaration. In addition to this, some parameters have to be set regarding setters/getters, as well as delegation.

To make the processor a working refactoring, a `MoveRefactoring` must be created with it as a parameter.

3.2.2 Shortcomings

This section is introduced naturally with a conclusion: The JDT refactoring implementations do not facilitate composition of refactorings. This section will try to explain why, and also try to identify other shortcomings of both the usability and readability of the JDT refactoring source code.

Absence of generics in Eclipse source code

This section is not only concerning the JDT refactoring API, but also large quantities of the Eclipse source code. The code shows a striking absence of the Java language feature of generics. It is hard to read a class' interface when methods return objects or takes parameters of raw types such as `List` or `Map`. This sometimes results in having to read a lot of source code to understand what is going on, instead of relying on the available interfaces. In addition, it results in a lot of ugly code, making the use of typecasting more of a rule than an exception.

Composite refactorings will not appear as atomic actions

When composing primitive refactorings from the JDT, it is not possible to make them appear as being executed as one change, but only as multiple small changes.

Missing flexibility from JDT refactorings. The JDT refactorings are not made with composition of refactorings in mind. When a JDT refactoring is executed, it assumes that all conditions for it to be applied successfully can be found by reading source files that have been persisted to disk. They can only operate on the actual source material, and not (in-memory) copies thereof. This constitutes a major disadvantage when trying to compose refactorings, since if an exception occurs in the middle of a sequence of refactorings, it can leave the project in a state where the composite refactoring was only partially executed. It makes it hard to discard the changes done without monitoring and consulting the undo manager, an approach that is not bullet proof.

Broken undo history. When designing a composite refactoring that is to be performed as a sequence of other refactorings, you would like it to appear as a single change to the workspace. This implies that you would also like to be able to undo all the changes done by the refactoring in a single step. This is not the way it appears when a sequence of JDT refactorings are executed. It leaves the undo history filled up with individual undo actions corresponding to every single JDT refactoring in the sequence. This problem is not trivial to handle in Eclipse (see section 4.3.4 on page 76).

Chapter 4

Source code organization and implementation details

4.1 Source code organization

All the parts of this master's project are under version control with Git.

The software written is organized as some Eclipse plugins. Writing a plugin is the natural way to utilize the API of Eclipse. This also makes it possible to provide a user interface to manually run operations on selections in program source code or whole entities, like methods or projects.

When writing a plugin for Eclipse, one has access to resources such as the current workspace, the open editor and the current selection.

The thesis work is contained in the following Eclipse projects:

no.uio.ifi.refaktor
: This is the main Eclipse plugin project, and contains all of the business logic for the plugin.

no.uio.ifi.refaktor.tests
: This project contains the tests for the main plugin.

no.uio.ifi.refaktor.examples
: Contains example code used in testing. It also contains code for managing this example code, such as creating an Eclipse project from it before a test run.

no.uio.ifi.refaktor.benchmark
: This project contains code for running search-based versions of the composite refactoring over selected Eclipse projects.

no.uio.ifi.refaktor.releng
: Contains the rmap, queries and target definitions needed by Buckminster on the Jenkins continuous integration server (see appendix B on page 113).

4.1.1 The no.uio.ifi.refaktor project

no.uio.ifi.refaktor.analyze

This package, and its sub-packages, contains code that is used for analyzing Java source code. The most important sub-packages are presented below.

no.uio.ifi.refaktor.analyze.analyzers
: This package contains source code analyzers. These are usually responsible for analyzing text selections or running specialized analyzers for different kinds of entities. Their structures are often hierarchical. This means that you have an analyzer for text selections, which in turn is utilized by an analyzer that analyzes all the selections of a method. Then there are analyzers for analyzing all the methods of a type, all the types of a compilation unit, all the compilation units of a package, and, at last, all of the packages in a project.

no.uio.ifi.refaktor.analyze.checkers
: A package containing checkers. The checkers are classes used to validate that a selection can be further analyzed and chosen as a candidate for a refactoring. Invalidating properties can be such as usage of inner classes or the need for multiple return values.

no.uio.ifi.refaktor.analyze.collectors
: This package contains the property collectors. Collectors are used to gather properties from a text selection. This is mostly properties regarding referenced names and their occurrences. These properties make up the basis for finding the candidates for a refactoring.

no.uio.ifi.refaktor.change

This package, and its sub-packages, contains functionality for manipulating source code.

no.uio.ifi.refaktor.change.changers
: This package contains source code changers. They are used to glue together the analysis of source code and the execution of the changes.

no.uio.ifi.refaktor.change.executors
: The executors that are responsible for making concrete changes are found in this package. They are mostly used to create and execute one or more JDT refactorings.

no.uio.ifi.refaktor.change.processors
: Contains a refactoring processor for the *Move Method* refactoring. The code is stolen and modified to fix a bug. The related bug is described in appendix A.2 on page 109.

no.uio.ifi.refaktor.handlers

This package contains handlers for the commands defined in the plugin manifest.

no.uio.ifi.refaktor.prefix

This package contains the `Prefix` type that is the data representation of the prefixes found by the `PrefixesCollector`. It also contains the prefix set for storing and working with prefixes.

no.uio.ifi.refaktor.statistics

This package contains statistics functionality. Its heart is the statistics aspect that is responsible for gathering statistics during the execution of the *Extract and Move Method* refactoring.

no.uio.ifi.refaktor.statistics.reports
: This package contains a simple framework for generating reports from the statistics data generated by the aspect. Currently, the only available report type is a simple text report.

no.uio.ifi.refaktor.textselection

This package contains the two custom text selections that are used extensively throughout the project. One of them is just a subclass of the other, to support the use of the memento pattern to optimize the memory usage during benchmarking.

no.uio.ifi.refaktor.debugging

The package contains a debug utility class. I addition to this, the package `no.uio.ifi.refaktor.utils.aspects` contains a couple of aspects used for debugging purposes.

no.uio.ifi.refaktor.utils

Utility package that contains all the functionality that has to do with parsing of source code. It also has utility classes for looking up handles to methods and types et cetera.

no.uio.ifi.refaktor.utils.caching
: This package contains the caching manager for compilation units, along with classes for different caching strategies.

no.uio.ifi.refaktor.utils.nullobjects
: Contains classes for creating different null objects. Most of the classes are used to represent null objects of different handle types. These null objects are returned from various utility classes instead of returning a `null` value when other values are not available.

4.2 Implementing source code analysis

This section gathers implementation details for the most important parts of the source code analysis for the *Extract and Move Method* refactoring.

4.2.1 Representing prefixes

This section shows the classes responsible for representing and working with prefixes.

The Prefix class

This class exists mainly for holding data about a prefix, such as the expression that the prefix represents and the occurrence count of the prefix within a selection. In addition to this, it has some functionality such as calculating its sub-prefixes and intersecting it with another prefix. The definition of the intersection between two prefixes is a prefix representing the longest common expression between the two.

The PrefixSet class

A prefix set holds elements of type `Prefix`. It is implemented with the help of a `HashMap`[1] and contains some typical set operations, but it does not implement the `Set`[2] interface, since the prefix set does not need all of the functionality a `Set` requires to be implemented. In addition, it needs some other functionality not found in the `Set` interface. So due to the relatively limited use of prefix sets, and that it almost always needs to be referenced as such, and not a `Set<Prefix>`, it remains as an ad hoc solution to a concrete problem.

There are two ways of adding prefixes to a `PrefixSet`. The first is through its `add` method. This works like one would expect from a set. It adds the prefix to the set if it does not already contain the prefix. The other way is to *register* the prefix with the set. When registering a prefix, if the set does not contain the prefix, it is just added. If the set contains the prefix, its count gets incremented. This is how the occurrence count is handled.

The prefix set also computes the set of prefixes that is not enclosing any prefixes of another set. This is kind of a set difference operation only for enclosing prefixes.

4.2.2 Property collectors

The prefixes and unfixes are found by property collectors[3]. A property collector is of the `ASTVisitor` type, and thus visits nodes of type `ASTNode` of the abstract syntax tree (see section 3.1.3 on page 57).

The PrefixesCollector. The `PrefixesCollector`[4] finds prefixes that makes up the basis for calculating move targets for the *Extract and Move Method* refactoring. It visits expression statements[5] and creates prefixes

[1]`java.util.HashMap`

[2]`java.util.Set`

[3]`no.uio.ifi.refaktor.extractors.collectors.PropertyCollector`

[4]`no.uio.ifi.refaktor.extractors.collectors.PrefixesCollector`

[5]`org.eclipse.jdt.core.dom.ExpressionStatement`

from its expressions in the case of method invocations. The prefixes found are registered with a prefix set, together with all its sub-prefixes.

The UnfixesCollector. The `UnfixesCollector`[1] finds unfixes within a selection. Its semantics is described in section 2.7 on page 46.

4.2.3 Checkers

The checkers are a range of classes that checks that text selections comply with certain criteria. All checkers operate under the assumption that the code they check is free from compilation errors. If a `Checker`[2] fails, it throws a `CheckerException`. The checkers are managed by the `LegalStatementsChecker`, which does not, in fact, implement the `Checker` interface. It does, however, run all the checkers registered with it, and reports that all statements are considered legal if no `CheckerException` is thrown. Many of the checkers either extends the `PropertyCollector` or utilizes one or more property collectors to verify different criteria. The checkers registered with the `LegalStatementsChecker` are described next. They are run in the order presented below.

The CallToProtectedOrPackagePrivateMethodChecker

This checker is used to check that at selection does not contain a call to a method that is protected or package-private. Such a method either has the access modifier `protected` or it has no access modifier.

The workings of the `CallToProtectedOrPackagePrivateMethod-Checker` is very simple. It looks for calls to methods that are either protected or package-private within the selection, and throws an `IllegalExpressionFoundException` if one is found.

The DoubleClassInstanceCreationChecker

The `DoubleClassInstanceCreationChecker` checks that there are no double class instance creations where the inner constructor call takes an argument that is built up using field references.

The checker visits all nodes of type `ClassInstanceCreation` within a selection. For all of these nodes, if its parent also is a class instance creation, it accepts a visitor that throws an `IllegalExpressionFoundException` if it encounters a name that is a field reference.

The InstantiationOfNonStaticInnerClassChecker

The `InstantiationOfNonStaticInnerClassChecker` checks that selections do not contain instantiations of non-static inner classes. The `MoveInstanceMethodProcessor` in Eclipse does not handle such instantiations gracefully when moving a method.

[1] `no.uio.ifi.refaktor.extractors.collectors.UnfixesCollector`
[2] `no.uio.ifi.refaktor.analyze.analyzers.Checker`

The EnclosingInstanceReferenceChecker

The purpose of this checker is to verify that the names in a text selection are not referencing any enclosing instances. In theory, the underlying problem could be solved in some situations, but our dependency on the `MoveInstanceMethodProcessor` prevents this.

The `EnclosingInstanceReferenceChecker`[1] is a modified version of the `EnclosingInstanceReferenceFinder`[2] from the `MoveInstanceMethod-Processor`. Wherever the `EnclosingInstanceReferenceFinder` would create a fatal error status, the checker will throw a `CheckerException`.

The checker works by first finding all of the enclosing types of a selection. Thereafter, it visits all the simple names of the selection to check that they are not references to variables or methods declared in any of the enclosing types. In addition, the checker visits `this`-expressions to verify that no such expressions are qualified with any name.

The ReturnStatementsChecker

The checker for return statements is meant to verify that a text selection is consistent regarding return statements.

If the selection is free from return statements, then the checker validates. So this is the first thing the checker investigates.

If the checker proceeds any further, it is because the selection contains one or more return statements. The next test is therefore to check if the last statement of the selection ends in either a return or a throw statement. The responsibility for checking that the last statement of the selection eventually ends in a return or throw statement, is put on the `LastStatementOfSelectionEndsInReturnOrThrowChecker`. For every node visited, if the node is a statement, it performs a test to see if the statement is a return, a throw or if it is an implicit return statement. If this is the case, no further checking is done. This checking is done in the `preVisit2` method (see section 3.1.3 on page 57). If the node is not of a type that is being handled by its type-specific visit method, the checker performs a simple test. If the node being visited is not the last statement of its parent that is also enclosed by the selection, an `IllegalStatementFoundException` is thrown. This ensures that all statements are taken care of, one way or the other. It also ensures that the checker is conservative in the way it checks for legality of the selection.

To examine if a statement is an implicit return statement, the checker first finds the last statement declared in its enclosing method. If this statement is the same as the one under investigation, it is considered an implicit return statement. If the statements are not the same, the checker does a search to see if the statement examined is also the last statement of the method that can be reached. This includes the last statement of a block statement, a labeled statement, a synchronized statement or a try

[1] `no.uio.ifi.refaktor.analyze.analyzers.EnclosingInstanceReferenceChecker`

[2] `org.eclipse.jdt.internal.corext.refactoring.structure.MoveInstanceMethod-Processor.EnclosingInstanceReferenceFinder`

statement, that in turn is the last statement enclosed by one of the statement types listed. This search goes through all the parents of a statement until a statement is found that is not one of the mentioned acceptable parent statements. If the search ends in a method declaration, then the statement is considered to be the last reachable statement of the method, and thus it is an implicit return statement.

There are two kinds of statements that are handled explicitly: If-statements and try-statements. Block, labeled and do-statements are handled by fall-through to the other two.

If-statements are handled explicitly by overriding their type-specific visit method. If the then-part does not contain any return or throw statements an **`IllegalStatementFoundException`** is thrown. If it does contain a return or throw, its else-part is checked. If the else-part is non-existent, or it does not contain any return or throw statements an exception is thrown. If no exception is thrown while visiting the if-statement, its children are visited.

A try-statement is checked very similar to an if-statement. Either its body or finally body must contain a return or throw. The same applies to its catch clauses. Failure to validate produces an **`IllegalStatementFoundException`**.

If the checker does not complain at any point, the selection is considered valid with respect to return statements.

The AmbiguousReturnValueChecker

This checker verifies that there are no ambiguous return values in a selection.

First, the checker needs to collect some data. Those data are the binding keys for all simple names that are assigned to within the selection, including variable declarations, but excluding fields. The checker also finds out whether a return statement is found in the selection or not. No further checks of return statements are needed, since, at this point, the selection is already checked for illegal return statements (see section 4.2.3 on the facing page).

After the binding keys of the assignees are collected, the checker searches the part of the enclosing method that is after the selection for references whose binding keys are among the collected keys. If more than one unique referral is found, or only one referral is found, but the selection also contains a return statement, we have a situation with an ambiguous return value, and an exception is thrown.

The IllegalStatementsChecker

This checker is designed to check for illegal statements.

Notice that labels in break and continue statements need some special treatment. Since a label does not have any binding information, we have to search upwards in the AST to find the **`LabeledStatement`** that corresponds to the label from the break or continue statement, and check that it is contained in the selection. If the break or continue statement does not have a label attached to it, it is checked that its innermost enclosing loop or switch statement (break statements only) also is contained in the selection.

4.2.4 Source code analyzers

The analyzers presented in this section are used to analyze source code for candidates to the *Extract and Move Method* refactoring. The **`ExtractAndMoveMethodAnalyzer`**[1] can be used to analyze a selection, while the **`SearchBasedExtractAndMoveMethodAnalyzer`**[2] analyzes all the text selection for a method.

The `ExtractAndMoveMethodAnalyzer`

This analyzer can perform analysis and precondition checking for an *Extract and Move Method* refactoring. First it checks whether a text selection is a valid selection or not, with respect to statement boundaries and that it actually contains any selections. Then it checks the legality of both extracting the selection and also moving it to another class. This checking of legality is performed by a range of checkers (see section 4.2.3 on page 69). If the selection is approved as legal, it is analyzed to find the presumably best target to move the extracted method to.

For finding the best suitable target the analyzer is using a **`PrefixesCollector`**[3] that collects all the possible candidate targets for the refactoring. All the non-candidates are found by an **`UnfixesCollector`**[4] that collects all the targets that will give some kind of error if used. (For details about the property collectors, see section 4.2.2 on page 68.) All prefixes (and unfixes) are represented by a **`Prefix`**[5], and they are collected into sets of prefixes. The safe prefixes are found by subtracting from the set of candidate prefixes the prefixes that are enclosing any of the unfixes. A prefix is enclosing an unfix if the unfix is in the set of its sub-prefixes. As an example, **`"a.b"`** is enclosing **`"a"`**, as is **`"a"`**. The safe prefixes is unified in a **`PrefixSet`**. If a prefix has only one occurrence, and is a simple expression, it is considered unsuitable as a move target. This occurs in statements such as **`"a.foo()"`**. For such statements it bares no meaning to extract and move them. It only generates an extra method and the calling of it.

The most suitable target for the refactoring is found by finding the prefix with the most occurrences. If two prefixes have the same occurrence count, but they differ in the number of segments, the one with most segments is chosen.

The `SearchBasedExtractAndMoveMethodAnalyzer`

This analyzer can be used for analyzing all the possible text selections of a method and then to choose the best result out of the analysis results

[1]**`no.uio.ifi.refaktor.analyze.analyzers.ExtractAndMoveMethodAnalyzer`**

[2]**`no.uio.ifi.refaktor.analyze.analyzers.SearchBasedExtractAndMoveMethodAnalyzer`**

[3]**`no.uio.ifi.refaktor.analyze.collectors.PrefixesCollector`**

[4]**`no.uio.ifi.refaktor.analyze.collectors.UnfixesCollector`**

[5]**`no.uio.ifi.refaktor.extractors.Prefix`**

that are, by the analyzer, considered to be the potential candidates for the *Extract and Move Method* refactoring.

Before the analyzer is able to work with the text selections of a method, it needs to generate them. To do this, it parses the method to obtain a `MethodDeclaration` for it (see section 3.1.2 on page 55). Then there is a statement lists creator that creates statement lists of the different groups of statements in the body of the method declaration. A text selections generator generates text selections of all the statement lists for the analyzer to work with.

The statement lists creator is responsible for generating lists of statements for all the possible nesting levels of statements in the method. The statement lists creator is implemented as an AST visitor (see section 3.1.3 on page 57). It generates lists of statements by visiting all the blocks in the method declaration and stores their statements in a collection of statement lists. In addition, it visits all of the other statements that can have a statement as a child, such as the different control structures and the labeled statement.

The switch statement is the only kind of statement that is not straight forward to obtain the child statements from. It stores all of its children in a flat list. Its switch case statements are included in this list. This means that there are potential statement lists between all of these case statements. The list of statements from a switch statement is therefore traversed, and the statements between the case statements are grouped as separate lists.

The highlighted part of listing 8 on page 39 shows an example of how the statement lists creator would separate a method body into lists of statements.

The text selections generator generates text selections for each list of statements from the statement lists creator. The generator generates a text selection for every sub-sequence of statements in a list. For a sequence of statements, the first statement and the last statement span out a text selection.

The text selections are calculated by traversing the statement list. There is a set of generated text selections. For each statement, there is created a temporary set of selections, in addition to a text selection based on the offset and length of the statement. This text selection is added to the temporary set. Then the new selection is added with every selection from the set of generated text selections. These new selections are added to the temporary set. Then the temporary set of selections is added to the set of generated text selections. The result of adding two text selections is a new text selection spanned out by the two addends.

Finding the candidate for the refactoring is done by analyzing all the generated text selections with an `ExtractAndMoveMethodAnalyzer` (see section 4.2.4 on the facing page). If the analyzer generates a useful result, an `ExtractAndMoveMethodCandidate` is created from it, which is kept in a list of potential candidates. If no candidates are found, the **`NoTargetFoundException`** is thrown.

Since only one of the candidates can be chosen, the analyzer must sort out which candidate to choose. The sorting is done by the static **`sort`** method of **`Collections`**. The comparison in this sorting is done by an **`ExtractAndMoveMethodCandidateComparator`**. The implementation used is the **`FavorNoUnfixesCandidateComparator`**. Its sort criteria are the same as in section 2.9 on page 50.

4.3 Composite refactoring implementations

This section will present how composite refactorings are implemented within the bounds of the Eclipse platform and the JDT.

4.3.1 A simple ad hoc model

As pointed out in section 3.2 on page 59, the Eclipse JDT refactoring model is not very well suited for making composite refactorings. Therefore, a simple model using changer objects (of type **`RefaktorChanger`**) is used as an abstraction layer on top of the existing Eclipse refactorings, instead of extending the **`Refactoring`**[1] class.

The use of an additional abstraction layer is a deliberate choice. It is due to the problem of creating a composite **`Change`**[2] that can handle text changes that interfere with each other. Thus, a **`RefaktorChanger`** may, or may not, take advantage of one or more existing refactorings, but it is always intended to make a change to the workspace.

The typical `RefaktorChanger`. The typical Refaktor changer class has two responsibilities: Checking preconditions and executing changes. This is not too different from the responsibilities of an LTK refactoring, with the distinction that a Refaktor changer also executes the change, while an LTK refactoring is only responsible for creating the object that can later be used to do that job.

Checking of preconditions is typically done by an **`Analyzer`**[3]. If the preconditions validate, the upcoming changes are executed by an **`Executor`**[4].

4.3.2 A simple Extract and Move Method refactoring

This section describes the implementation of a simple refactoring, that for a given text selection will analyze it and perform the *Extract and Move Method* refactoring if a suitable move target is found within the selection.

[1]`org.eclipse.ltk.core.refactoring.Refactoring`
[2]`org.eclipse.ltk.core.refactoring.Change`
[3]`no.uio.ifi.refaktor.analyze.analyzers.Analyzer`
[4]`no.uio.ifi.refaktor.change.executors.Executor`

The `ExtractAndMoveMethodChanger`. This changer[1] is a subclass of the class **`RefaktorChanger`**[2]. It is responsible for analyzing and finding the best target for, and also executing, an *Extract and Move Method* refactoring. This particular changer is the one of my changers that is closest to being a true LTK refactoring. It can be reworked to be one if the problems with overlapping changes are resolved.

The changer requires a text selection and the name of the new method, otherwise a method name will be generated. The selection has to be of the type **`CompilationUnitTextSelection`**[3]. This class is a custom extension to **`TextSelection`**[4], that in addition to the basic offset, length and similar methods, also carry an instance of the underlying compilation unit handle for the selection.

The analysis and precondition checking for this changer is done by an **`ExtractAndMoveMethodAnalyzer`** (see section 4.2.4 on page 72), and the execution is done by an **`ExtractAndMoveMethodExecutor`**.

The `ExtractAndMoveMethodExecutor`. If the analysis finds a possible target for the composite refactoring, it is executed by an **`ExtractAndMoveMethodExecutor`**[5]. It is composed of the two executors known as **`ExtractMethodRefactoringExecutor`**[6] and **`MoveMethodRefactoringExecutor`**[7]. The **`ExtractAndMoveMethodExecutor`** is responsible for gluing the two together by feeding the **`MoveMethodRefactoringExecutor`** with the resources needed after executing the *Extract Method* refactoring.

The `ExtractMethodRefactoringExecutor`. This executor is responsible for creating and executing an instance of the **`ExtractMethodRefactoring`** class. It is also responsible for collecting some post execution resources that can be used to find the method handle for the extracted method, as well as information about its parameters, including the variable they originated from.

The `MoveMethodRefactoringExecutor`. This executor is responsible for creating and executing an instance of the **`MoveRefactoring`**. The move refactoring is a processor-based refactoring, and for the *Move Method* refactoring it is the modified version of the **`MoveInstanceMethodProcessor`** that is used (see appendix A.2 on page 109).

The handle for the method to be moved is found on the basis of the information gathered after the execution of the *Extract Method* refactoring. The only information the **`ExtractMethodRefactoring`** is sharing after its execution, regarding finding the method handle, is the textual representation

[1] `no.uio.ifi.refaktor.changers.ExtractAndMoveMethodChanger`
[2] `no.uio.ifi.refaktor.changers.RefaktorChanger`
[3] `no.uio.ifi.refaktor.utils.CompilationUnitTextSelection`
[4] `org.eclipse.jface.text.TextSelection`
[5] `no.uio.ifi.refaktor.change.executors.ExtractAndMoveMethodExecutor`
[6] `no.uio.ifi.refaktor.change.executors.ExtractMethodRefactoringExecutor`
[7] `no.uio.ifi.refaktor.change.executors.MoveMethodRefactoringExecutor`

of the new method signature. Therefore it must be parsed, the strings for types of the parameters must be found and translated to a form that can be used to look up the method handle from its type handle. They have to be on the unresolved form. The name for the type is found from the original selection, since an extracted method must end up in the same type as the originating method.

When analyzing a selection prior to performing the *Extract Method* refactoring, a target is chosen. It has to be a variable binding, so it is either a field or a local variable/parameter. If the target is a field, it can be used with the **`MoveInstanceMethodProcessor`** as it is, since the extracted method still is in its scope. But if the target is local to the originating method, the target that is to be used for the processor must be among its parameters. Thus the target must be found among the extracted method's parameters. This is done by finding the parameter information object that corresponds to the parameter that was declared on basis of the original target's variable when the method was extracted. (The extracted method must take one such parameter for each local variable that is declared outside the selection that is extracted.) To match the original target with the correct parameter information object, the key for the information object is compared to the key from the original target's binding. The source code must then be parsed to find the method declaration for the extracted method. The new target must be found by searching through the parameters of the declaration and choose the one that has the same type as the old binding from the parameter information object, as well as the same name that is provided by the parameter information object.

4.3.3 An on-demand search-based Extract and Move Method refactoring

The **`SearchBasedExtractAndMoveMethodChanger`**[1] is a changer whose purpose is to automatically analyze a method, and execute the *Extract and Move Method* refactoring on it if it is a suitable candidate for the refactoring.

First, the **`SearchBasedExtractAndMoveMethodAnalyzer`** is used to analyze the method. If the method is found to be a candidate, the result from the analysis is fed to the **`ExtractAndMoveMethodExecutor`**, whose job is to execute the refactoring (see section 4.3.2 on the previous page).

4.3.4 Hacking the refactoring undo history

As an attempt to make multiple subsequent changes to the workspace appear as a single action (i.e. make the undo changes appear as such), I tried to alter the undo changes[2] in the history of the refactorings.

My first impulse was to remove the, in this case, last two undo changes from the undo manager[3] for the Eclipse refactorings, and then add them to a

[1]`no.uio.ifi.refaktor.change.changers.SearchBasedExtractAndMoveMethodChanger`
[2]`org.eclipse.ltk.core.refactoring.Change`
[3]`org.eclipse.ltk.core.refactoring.IUndoManager`

composite change[1] that could be added back to the manager. The interface of the undo manager does not offer a way to remove/pop the last added undo change, so a possible solution could be to decorate [Gam+95] the undo manager, to intercept and collect the undo changes before delegating to the **`addUndo`** method[2] of the manager. Instead of giving it the intended undo change, a null change could be given to prevent it from making any changes if run. Then one could let the collected undo changes form a composite change to be added to the manager.

There is a technical challenge with this approach, and it relates to the undo manager, and the concrete implementation **`UndoManager2`**[3]. This implementation is designed in a way that it is not possible to just add an undo change, you have to do it in the context of an active operation[4]. One could imagine that it might be possible to trick the undo manager into believing that you are doing a real change, by executing a refactoring that is returning a kind of null change that is returning our composite change of undo refactorings when it is performed. But this is not the way to go.

Apart from the technical problems with this solution, there is a functional problem: If it all had worked out as planned, this would leave the undo history in a dirty state, with multiple empty undo operations corresponding to each of the sequentially executed refactoring operations, followed by a composite undo change corresponding to an empty change of the workspace for rounding of our composite refactoring. The solution to this particular problem could be to intercept the registration of the intermediate changes in the undo manager, and only register the last empty change.

Unfortunately, not everything works as desired with this solution. The grouping of the undo changes into the composite change does not make the undo operation appear as an atomic operation. The undo operation is still split up into separate undo actions, corresponding to the changes done by their originating refactorings. And in addition, the undo actions have to be performed separately in all the editors involved. This makes it no solution at all, but a step toward something worse.

There might be a solution to this problem, but it remains to be found. The design of the refactoring undo management is partly to be blamed for this, as it is too complex to be easily manipulated.

4.4 Benchmarking

This part of the master's project is located in the Eclipse project **`no.uio.ifi.refaktor.benchmark`**. The purpose of it is to run the equivalent of the **`SearchBasedExtractAndMoveMethodChanger`** (see section 4.3.3 on the facing page) over a larger software project, both to test its robustness but also its effect on different software metrics.

[1]`org.eclipse.ltk.core.refactoring.CompositeChange`
[2]`org.eclipse.ltk.core.refactoring.IUndoManager#addUndo()`
[3]`org.eclipse.ltk.internal.core.refactoring.UndoManager2`
[4]`org.eclipse.core.commands.operations.TriggeredOperations`

4.4.1 The benchmark setup

The benchmark itself is set up as a JUnit test case. This is a convenient setup, and utilizes the JUnit Plugin Test Launcher. This provides us with a fully functional Eclipse workbench. Most importantly, this gives us access to the Java Model of Eclipse (see section 3.1.1 on page 53).

The ProjectImporter

The Java project that is going to be used as the data for the benchmark, must be imported into the JUnit workspace. This is done by the `ProjectImporter`[1]. The importer requires the absolute path to the project description file. This file is named `.project` and is located at the root of the project directory.

The project description is loaded to find the name of the project to be imported. The project that shall be the destination for the import is created in the workspace, on the base of the name from the description. Then an import operation is created, based on both the source and destination information. The import operation is run to perform the import.

I have found no simple API call to accomplish what the importer does, which tells me that it may not be too many people performing this particular action. The solution to the problem was found on Stack Overflow[2]. It contains enough dirty details to be considered inconvenient to use, if not wrapping it in a class like my `ProjectImporter`. One would probably have to delve into the source code for the import wizard to find out how the import operation works, if no one had already done it.

4.4.2 Statistics

Statistics for the analysis and changes is captured by the `StatisticsAspect`[3]. This an *aspect* written in AspectJ.

AspectJ

AspectJ is an extension to the Java language, and facilitates combining aspect-oriented programming with the object-oriented programming in Java.

Aspect-oriented programming is a programming paradigm that is meant to isolate so-called *cross-cutting concerns* into their own modules. These cross-cutting concerns are functionalities that span over multiple classes, but may not belong naturally in any of them. It can be functionality that does not concern the business logic of an application, and thus may be a burden when entangled with parts of the source code it does not really belong to. Examples include logging, debugging, optimization and security.

Aspects are interacting with other modules by defining advices. The concept of an *advice* is known from both aspect-oriented and functional programming. It is a function that modifies another function when the

[1]`no.uio.ifi.refaktor.benchmark.ProjectImporter`
[2]https://stackoverflow.com/questions/12401297
[3]`no.uio.ifi.refaktor.aspects.StatisticsAspect`

latter is run. An advice in AspectJ is somewhat similar to a method in Java. It is meant to alter the behavior of other methods, and contains a body that is executed when it is applied.

An advice can be applied at a defined *pointcut*. A pointcut picks out one or more *join points*. A join point is a well-defined point in the execution of a program. It can occur when calling a method defined for a particular class, when calling all methods with the same name, accessing/assigning to a particular field of a given class and so on. An advice can be declared to run both before, after returning from a pointcut, when there is thrown an exception in the pointcut or after the pointcut either returns or throws an exception. In addition to picking out join points, a pointcut can also bind variables from its context, so they can be accessed in the body of an advice. An example of a pointcut and an advice is found in listing 13.

```
pointcut methodAnalyze(
  SearchBasedExtractAndMoveMethodAnalyzer analyzer) :
    call(* SearchBasedExtractAndMoveMethodAnalyzer.analyze())
      && target(analyzer);

after(SearchBasedExtractAndMoveMethodAnalyzer analyzer) :
    methodAnalyze(analyzer) {
  statistics.methodCount++;
  debugPrintMethodAnalysisProgress(analyzer.method);
}
```

Listing 13: An example of a pointcut named **`methodAnalyze`**, and an advice defined to be applied after it has occurred.

The Statistics class

The statistics aspect stores statistical information in an object of type **`Statistics`**. As of now, the aspect needs to be initialized at the point in time where it is desired that it starts its data gathering. At any point in time the statistics aspect can be queried for a snapshot of the current statistics.

The **`Statistics`** class also includes functionality for generating a report of its gathered statistics. The report can be given either as a string or it can be written to a file.

Advices

The statistics aspect contains advices for gathering statistical data from different parts of the benchmarking process. It captures statistics from both the analysis part and the execution part of the composite *Extract and Move Method* refactoring.

For the analysis part, there are advices to count the number of text selections analyzed and the number of methods, types, compilation units and packages analyzed. There are also advices that counts for how many of the methods there are found a selection that is a candidate for the refactoring, and for how many methods there are not.

There exists advices for counting both the successful and unsuccessful executions of all the refactorings. Both for the *Extract Method* and *Move Method* refactorings in isolation, as well as for the combination of them.

4.4.3 Optimizations

When looking for possible optimizations for the benchmarking process, I used the VisualVM profiler for the Java Virtual Machine to both profile the application and also to make memory dumps of its heap.

Caching

When profiling the benchmark process before making any optimizations, it early became apparent that the parsing of source code was a place to direct attention toward. This discovery was done when only *analyzing* source code, before trying to do any *manipulation* of it. Caching of the parsed ASTs seemed like the best way to save some time, as expected. With only a simple cache of the most recently used AST, the analysis time was speeded up by a factor of around 20. This number depends a little upon which type of system the analysis is run.

The caching is managed by a cache manager, which now, by default, utilizes the not so well known feature of Java called a *soft reference*. Soft references are best explained in the context of weak references. A *weak reference* is a reference to an object instance that is only guaranteed to persist as long as there is a *strong reference* or a soft reference referring the same object. If no such reference is found, its referred object is garbage collected. A strong reference is basically the same as a regular Java reference. A soft reference has the same guarantees as a week reference when it comes to its relation to strong references, but it is not necessarily garbage collected if there are no strong references to it. A soft reference *may* reside in memory as long as the JVM has enough free memory in the heap. A soft reference will therefore usually perform better than a weak reference when used for simple caching and similar tasks. The way to use a soft/weak reference is to ask it for its referent. The return value then has to be tested to check that it is not **`null`**. For the basic usage of soft references, see listing 14. For a more thorough explanation of weak references in general, see [Nic06].

The cache based on soft references has no limit for how many ASTs it caches. It is generally not advisable to keep references to ASTs for prolonged periods of time, since they are expensive structures to hold on to. For regular plugin development, Eclipse recommends not creating more than one AST at a time to limit memory consumption. Since the benchmarking has nothing to do with user experience, and throughput is everything, these advices are intentionally ignored. This means that during the benchmarking process, the target Eclipse application may very well work close to its memory limit for the heap space for long periods during the benchmark.

```java
// Strong reference
Object strongRef = new Object();

// Soft reference
SoftReference<Object> softRef =
    new SoftReference<Object>(new Object());

// Using the soft reference
Object obj = softRef.get();
if (obj != null) {
    // Use object here
}
```

Listing 14: Showing the basic usage of soft references. Weak references are used the same way. (The references are part of the `java.lang.ref` package.)

Candidates stored as mementos

When performing large scale analysis of source code for finding candidates to the *Extract and Move Method* refactoring, memory is an issue. One of the inputs to the refactoring is a variable binding. This variable binding indirectly retains a whole AST. Since ASTs are large structures, this quickly leads to an `OutOfMemoryError` if trying to analyze a large project without optimizing how we store the candidates' data. This means that the JVM cannot allocate more memory for our benchmark, and it exits disgracefully.

A possible solution could be to just allow the JVM to allocate even more memory, but this is not a dependable solution. The allocated memory could easily supersede the physical memory of a machine, which would make the benchmark go really slow.

Thus, the candidates' data must be stored in another format. Therefore, we use the memento pattern to store variable binding information. This is done in a way that makes it possible to retrieve a variable binding at a later point. The data that is stored to achieve this, is the key to the original variable binding. In addition to the key, we know which method and text selection the variable is referenced in, so that we can find it by parsing the source code and search for it when it is needed.

4.4.4 Handling failures

Failures during the benchmarking process are logged and then ignored. The failures are represented in the statistics gathered.

Chapter 5

Case studies

In this chapter I will present two case studies. This is done to give an impression of how the search-based *Extract and Move Method* refactoring performs when giving it a larger project to take on. I will try to answer where it lacks, in terms of completeness, as well as showing its effect on refactored source code.

The first and primary case, is refactoring source code from the Eclipse JDT UI project. The project is chosen because it is a well-known open-source project, still in development, with a large code base that is written by many different people over several years. The code is installed in a large number of Eclipse applications worldwide, and many other projects build on the Eclipse platform. For a long time, it was even the official IDE for Android development. All this means that Eclipse must be seen as a good representative for professionally written Java source code. It is also the home for most of the JDT refactoring code.

For the second case, the *Extract and Move Method* refactoring is fed the `no.uio.ifi.refaktor` project. This is done as a variation of the "dogfooding" methodology.

5.1 The tools

For conducting these experiments, three software tools are used. Two of the tools both use Eclipse as their platform. The first is our own tool, described in section 4.4 on page 77, written to be able to run the *Extract and Move Method* refactoring as a batch process. It analyzes and refactors all the methods of a project in sequence. The second is JUnit, which is used for running the project's own unit tests on the target code both before and after it is refactored. The last tool that is used is a code quality management tool, called SonarQube. It can be used to perform different tasks for assuring code quality, but we are only going to take advantage of one of its main features, namely quality profiles.

A quality profile is used to define a set of coding rules that a project is supposed to comply with. Failure to following these rules will be recorded as so-called "issues", marked as having one of several degrees of severities, ranging from "info" to "blocker", where the latter one is the most severe.

The measurements done for these case studies are therefore not presented as fine-grained software metrics results, but rather as the number of issues for each defined rule.

In its analysis, SonarQube discriminates between functions and accessors. Accessors are methods that are recognized as setters or getters.

In addition to the coding rules defined through quality profiles, SonarQube calculates the complexity of source code. The metric that is used is cyclomatic complexity, developed by Thomas J. McCabe in 1976 [McC76]. In this metric, functions have an initial complexity of 1, and whenever the control flow of a function splits, the complexity increases by one[1]. Accessors are not counted in the complexity analysis.

The specifications for the computer used during the experiments are shown in table 5.1.

Table 5.1: Specifications for experiment computer.

Hardware	
Model	Lenovo ThinkPad Edge S430
Processor	Intel® Core™ i5-3210M (2.5 GHz/3.1 GHz (turbo), 2 cores, 4 threads, 3 MB Cache)
Memory	8 GB DDR3 1600 MHz
Storage	500 GB HDD (7200 RPM) + 16 GB SSD Cache for Lenovo Hard Disk Drive Performance Booster
Operating system	
Distribution	Ubuntu 12.10
Kernel	Linux 3.5.0-49-generic (x86_64)

5.2 The SonarQube quality profile

The quality profile that is used with SonarQube in these case studies has got the name IFI Refaktor Case Study (version 6). The rules defined in the profile are chosen because they are the available rules found in SonarQube that measures complexity and coupling. Now follows a description of the rules in the quality profile. The values that are set for these rules are listed in table 5.2.

Avoid too complex class is a rule that measures cyclomatic complexity for every statement in the body of a class, except for setters and getters. The threshold value set is its default value of 200.

Classes should not be coupled to too many other classes is a rule that measures how many other classes a class depends upon. It does

[1] http://docs.codehaus.org/display/SONAR/Metric+definitions

not count the dependencies of nested classes. It is meant to promote the Single Responsibility Principle. The metric for the rule resembles the CBO metric that is described in section 1.3.5 on page 29, but is only considering outgoing dependencies. The max value for the rule is chosen on the basis of an empirical study by Raed Shatnawi, which concludes that the number 9 is the most useful threshold for the CBO metric [Sha10]. This study is also performed on Eclipse source code, so this threshold value should be particularly well suited for the Eclipse JDT UI case in this chapter.

Control flow statements ... should not be nested too deeply is a rule that is meant to counter “Spaghetti code”. It measures the nesting level of *if*, *for*, *while*, *switch* and *try* statements. The nesting levels start at 1. The max value set is its default value of 3.

Methods should not be too complex is a rule that measures cyclomatic complexity the same way as the “Avoid too complex class” rule. The max value used is 10, which “seems like a reasonable, but not magical, upper limit“ [McC76].

Methods should not have too many lines is a rule that simply measures the number of lines in methods. A threshold value of 20 is used for this metric. This is based on my own subjective opinions, as the default value of 100 describes method bodies that do not even fit on most screens.

NPath Complexity is a rule that measures the number of possible execution paths through a function. The value used is the default value of 200, which seems like a recognized threshold for this metric.

Too many methods is a rule that measures the number of methods in a class. The threshold value used is the default value of 10.

Table 5.2: The IFI Refaktor Case Study quality profile (version 6).

Rule	Max value
Avoid too complex class	200
Classes should not be coupled to too many other classes (Single Responsibility Principle)	9
Control flow statements ... should not be nested too deeply	3
Methods should not be too complex	10
Methods should not have too many lines	20
NPath Complexity	200
Too many methods	10

5.3 The input

A precondition for the source code that is going to be the target for a series of *Extract and Move Method* refactorings, is that it is organized as an Eclipse project. It is also assumed that the code is free from compilation errors.

5.4 The experiment

For a given project, the first job that is done, is to refactor its source code. The refactoring batch job produces three things: The refactored project, statistics gathered during the execution of the series of refactorings, and an error log describing any errors happening during this execution. See section 4.4 on page 77 for more information about how the refactorings are performed.

After the refactoring process is done, the before- and after-code is analyzed with SonarQube. The analysis results are then stored in a database and displayed through a SonarQube server with a web interface.

The before- and after-code is also tested with their own unit tests. This is done to discover any changes in the semantic behavior of the refactored code, within the limits of these tests.

5.5 Case 1: The Eclipse JDT UI project

This case is the ultimate test for our *Extract and Move Method* refactoring. The target source code is massive. With its over 300,000 lines of code[1] and more than 25,000 methods, it is a formidable task to perform automated changes on it. There should be plenty of situations where things can go wrong.

I will start by presenting some statistics from the refactoring execution, before I pick apart the SonarQube analysis and conclude by commenting on the results from the unit tests. The configuration for the experiment is specified in table 5.3.

5.5.1 Statistics

The statistics gathered during the refactoring execution is presented in table 5.4 on page 94.

Execution time

I consider the total execution time of approximately 1.5 hours, on a mid-level laptop computer, as being acceptable. It clearly makes the batch process unsuitable for doing any on-demand analysis or changes, but it is good enough for running periodic jobs, like over-night analysis. In comparison, the

[1]For all uses of “lines of code” we follow the definition from SonarQube. LOC = the number of physical lines containing a character which is neither whitespace or part of a comment.

Table 5.3: Configuration for Case 1.

Benchmark data	
Launch configuration	CaseStudy.launch
Project	no.uio.ifi.refaktor.benchmark
Repository	gitolite@git.uio.no:ifi-stolz-refaktor
Commit	43c16c04520746edd75f8dc2a1935781d3d9de6c
Input data	
Project	org.eclipse.jdt.ui
Repository	git://git.eclipse.org/gitroot/jdt/eclipse.jdt.ui.git
Commit	f218388fea6d4ec1da7ce22432726c244888bb6b
Branch	R3_8_maintenance
Tests suites	org.eclipse.jdt.ui.tests.AutomatedSuite, org.eclipse.jdt.ui.tests.refactoring.all.-AllAllRefactoringTests

SonarQube analysis for the same project consumes about the same amount of time.

As the statistics show, 75% of the total time goes into making the actual code changes. The time consumers are here the primitive *Extract Method* and *Move Method* refactorings. Included in the change time is the parsing and precondition checking done by these refactorings, as well as textual changes done to files on disk. All this parsing and disk access is time-consuming, and constitutes a large part of the change time.

The pure analysis time, which is the time used on finding suitable refactoring candidates, only makes up for 15% of the total time consumed. This includes analyzing almost 600,000 text selections, while the number of attempted executions of the *Extract and Move Method* refactoring is only about 2,500. So the number of executed primitive refactorings is approximately 5,000. Assuming the time used on miscellaneous tasks are used mostly for parsing source code for the analysis, we can say that the time used for analyzing code is at most 25% of the total time. This means that for every primitive refactoring executed, we can analyze about 360 text selections. So, with an average of about 21 text selections per method, it is reasonable to say that we can analyze over 15 methods in the time it takes to perform a primitive refactoring.

Refactoring candidates

Out of the 27,667 methods that were analyzed, 2,552 methods contained selections that were considered candidates for the *Extract and Move Method* refactoring. This is roughly 9% of the methods in the project. These 9% of the methods had on average 14.4 text selections that were considered possible refactoring candidates.

Executed refactorings

2,469 out of 2,552 attempts on executing the *Extract and Move Method* refactoring were successful, giving a success rate of 96.7%. The failure rate of 3.3% stems from situations where the analysis finds a candidate selection, but the change execution fails. This failure could be an exception that was thrown, and the refactoring aborts. It could also be the precondition checking for one of the primitive refactorings that gives us an error status, meaning that if the refactoring proceeds, the code will contain compilation errors afterwards, forcing the composite refactoring to abort.

Out of the 2,552 *Extract Method* refactorings that were attempted executed, 69 of them failed. This gives a failure rate of 2.7% for the primitive refactoring. In comparison, the *Move Method* refactoring had a failure rate of 0.6 % of the 2,483 attempts on the refactoring.

If we also take into account that the pre-refactoring analysis is incomplete (see section 2.4.7 on page 45), the failure rates for the refactorings are not that bad.

5.5.2 SonarQube analysis

Results from the SonarQube analysis are shown in table 5.5 on page 95.

Diversity in the number of entities analyzed

The analysis performed by SonarQube is reporting fewer methods than found by the pre-refactoring analysis. SonarQube discriminates between functions (methods) and accessors, so the 1,296 accessors play a part in this calculation. SonarQube also has the same definition as our plugin when it comes to how a class is defined. Therefore it seems like SonarQube misses 277 classes that our plugin handles. This can explain why the SonarQube report differs from our numbers by approximately 2,500 methods.

Complexity

For all complexity rules that works on the method level, the number of issues decreases with between 3.1% and 6.5% from before to after the refactoring. The average complexity of a method decreases from 3.6 to 3.3, which is an improvement of about 8.3%. So, on the method level, the refactoring must be said to have a slightly positive impact. This is due to the extraction of a lot of methods, making the average method size smaller.

The improvement in complexity on the method level is somewhat traded for complexity on the class level. The complexity per class metric is worsened by 3% from before to after. The issues for the "Too many methods" rule also increases by 14.5%. These numbers indicate that the refactoring makes quite a lot of the classes a little more complex overall. This is the expected outcome, since the *Extract and Move Method* refactoring introduces almost 2,500 new methods into the project.

The only number that can save the refactoring's impact on complexity on the class level, is the "Avoid too complex class" rule. It improves with 2.5%, thus indicating that the complexity is moderately better distributed between the classes after the refactoring than before.

Coupling

One of the hopes when starting this project, was to be able to make a refactoring that could lower the coupling between classes. Better complexity at the method level is a not very unexpected byproduct of dividing methods into smaller parts. Lowering the coupling on the other hand, is a far greater task. This is also reflected in the results for the only coupling rule defined in the SonarQube quality profile, namely the "Classes should not be coupled to too many other classes (Single Responsibility Principle)" rule.

The number of issues for the coupling rule is 1,098 before the refactoring, and 1,199 afterwards. This is an increase in issues of 9.2%. These numbers can be interpreted two ways. The first possibility is that our assumptions are wrong, and that increasing indirection does not decrease coupling between classes. The other possibility is that our analysis and choices of candidate text selections are not good enough. I vote for the second possibility. (Voting against the public opinion may also be a little bold.)

An example of what makes the number of coupling issues grow

Listing 15 shows a portion of the class **`ShowActionGroup`**[1] from the JDT UI project before it is refactored with the search-based *Extract and Move Method* refactoring. Before the refactoring, the **`ShowActionGroup`** class has 12 outgoing dependencies (reported by SonarQube).

```java
public class ShowActionGroup extends ActionGroup {
  /* ... */
  private void initialize(IWorkbenchSite site,
                          boolean isJavaEditor) {
    fSite= site;
    ISelectionProvider provider= fSite.getSelectionProvider();
    ISelection selection= provider.getSelection();
    fShowInPackagesViewAction.update(selection);
    if (!isJavaEditor) {
      provider.addSelectionChangedListener(
                               fShowInPackagesViewAction);
    }
  }
}
```

Listing 15: Portion of the **`ShowActionGroup`** class before refactoring.

During the benchmark process, the search-based *Extract and Move*

[1]`org.eclipse.jdt.ui.actions.ShowActionGroup`

Method refactoring extracts the lines 6 to 12 of the code in listing 15 on the preceding page, and moves the new method to the move target, which is the field **fShowInPackagesViewAction** with type **ShowInPackageViewAction**[1]. The result is shown in listing 16.

```
public class ShowActionGroup extends ActionGroup {
  /* ... */
  private void initialize(IWorkbenchSite site,
                          boolean isJavaEditor) {
    fSite= site;
    fShowInPackagesViewAction.generated_8019497110545412081(
                                          this, isJavaEditor);
  }
}
```

```
public class ShowInPackageViewAction
        extends SelectionDispatchAction {
  /* ... */
  public void generated_8019497110545412081(
      ShowActionGroup showactiongroup, boolean isJavaEditor) {
    ISelectionProvider provider=
                showactiongroup.fSite.getSelectionProvider();
    ISelection selection= provider.getSelection();
    update(selection);
    if (!isJavaEditor) {
      provider.addSelectionChangedListener(this);
    }
  }
}
```

Listing 16: Portions of the classes **ShowActionGroup** and **ShowInPackageViewAction** after refactoring.

After the refactoring, the **ShowActionGroup** has only 11 outgoing dependencies. It no longer depends on the **ISelection** type. So our refactoring managed to get rid of one dependency, which is exactly what we wanted. The only problem is, that now the **ShowInPackageViewAction** class has got two new dependencies, in the **ISelectionProvider** and the **ISelection** types. The bottom line is that we eliminated one dependency, but introduced two more, ending up with a program that has more dependencies now than when we started out.

What can happen in many situations where the *Extract and Move Method* refactoring is performed, is that the *Move Method* refactoring "drags" with it references to classes that are unknown to the method destination. If the refactoring happens to be so lucky that it removes a dependency from one class, it might as well introduce a couple of new dependencies to another class, as shown in the previous example. In those

[1]**org.eclipse.jdt.ui.actions.ShowInPackageViewAction**

situations where a destination class does not know about the originating class of a moved method, the *Move Method* refactoring most certainly will introduce a dependency. This is because there is a bug[1] in the refactoring, making it pass an instance of the originating class as a reference to the moved method, regardless of whether the reference is used in the method body or not.

There is also the possibility that the heuristics used to find candidate text selections are not good enough. There is work to be done with fine-tuning the heuristics and to complete the analysis part of this project.

Totals

On the bright side, the total number of issues is lower after the refactoring than it was before. Before the refactoring, the total number of issues was 8,270, opposed to 8,155 after. This is an improvement of 1.4%.

The down side is that SonarQube shows that the total cyclomatic complexity has increased by 2.9%, and that the (more questionable) "technical debt" has increased from 1,003.4 to 1,032.7 days, also a deterioration of 2.9%. Although these numbers are similar, no correlation has been found between them.

5.5.3 Unit tests

The tests that have been run for the Eclipse JDT UI project, are the test suites specified as the main test suites on the JDT UI wiki page on how to contribute to the project[2]. The results from these tests are shown in table 5.6 on page 96.

Before the refactoring

Running the tests for the before-code of Eclipse JDT UI yielded 4 errors and 3 failures for the **`AutomatedSuite`** test suite (2,007 test cases), and 2 errors and 3 failures for the **`AllAllRefactoringTests`** test suite (3,816 test cases).

After the refactoring

For the after-code of the Eclipse JDT UI project, Eclipse reports that the project contains 322 compilation errors, and a lot of other errors that follow from these. All of the errors are caused by the *Extract and Move Method* refactoring. Had these errors originated from only one bug, it would not have been much of a problem, but this is not the case. By only looking at some random compilation problems in the refactored code, I came up with at least four different bugs that caused those problems. I then stopped looking for more, since some of the bugs would take more time to fix than I could justify using on them at this point.

[1]Eclipse Bug 228635 - [move method] unnecessary reference to source

[2]https://wiki.eclipse.org/JDT_UI/How_to_Contribute#Unit_Testing

One thing that can be said in my defense, is that all the compilation errors could have been avoided if the types of situations that cause them were properly handled by the primitive refactorings, which again are supplied by the Eclipse JDT UI project. All four bugs that I mentioned before are weaknesses of the *Move Method* refactoring. If the primitive refactorings had detected the up-coming errors in their precondition checking phase, the refactorings would have been aborted, since this is how the *Extract and Move Method* refactoring handles such situations. This shows that it is not safe to completely rely upon the primitive refactorings to save us if our own pre-refactoring analysis fails to detect that a compilation error will be introduced. A problem is that the source code analysis done by both the JDT refactorings and our own tool is incomplete.

Of course, taking into account all possible situations that could lead to compilation errors is an immense task. A complete analysis of these situations is too big of a problem for this master's project to solve. Looking at it now, this comes as no surprise, since the task is obviously also too big for the creators of the primitive *Move Method* refactoring.

Considering all these problems, it is difficult to know how to interpret the unit test results from after refactoring the Eclipse JDT UI. The **`AutomatedSuite`** reported 565 errors and 5 failures, which means that 1437, or 71.6%, of the tests still passed. Three of the failures were the same as reported before the refactoring took place, so two of them are new. For these two cases it is not immediately apparent what makes them behave differently. The program is so complex that to analyze it to find this out, we might need more powerful methods than just manually analyzing its source code. This is somewhat characteristic for imperative programming: The programs are often hard to analyze and understand.

For the **`AllAllRefactoringTests`** test suite, the three failures are gone, but the two errors have grown to 2,257 errors. I will not try to analyze those errors.

What I can say at this point, is that it is likely that the *Extract and Move Method* refactoring has introduced some unintentional behavioral changes. Let us say that the refactoring introduces at least two behavior-altering changes for every 2,500 executions. More than that is difficult to say about the behavior-preserving properties of the *Extract and Move Method* refactoring, at this point. What is clear, is that it would benefit from a strategy for making it perform refactoring in a safer manner.

5.5.4 Conclusions

After automatically analyzing and executing the *Extract and Move Method* refactoring for all the methods in the Eclipse JDT UI project, the results do not look that promising. For this case, the refactoring seems almost unusable as it is now. The error rate and measurements tell us this.

The refactoring makes the code a little less complex at the method level. But this is merely a side effect of extracting methods. When it comes to the overall complexity, it is increased, although it is slightly better spread among the classes.

The pre-refactoring analysis of the *Extract and Move Method* refactoring, is currently not complete enough to make the refactoring useful. It introduces too many errors in the code, and the code may change its behavior. It also remains to prove that large-scale refactoring with it can decrease the overall coupling between classes, although individual examples exist (see section 5.5.2 on page 89).

On the bright side, the performance of the refactoring process is not that bad. It shows that it is possible to make a tool the way we do, if we can make the tool do something useful. As long as the analysis phase is not going to involve anything that uses too much disk access, a lot of analysis can be done in a reasonable amount of time.

The time used on performing the actual changes excludes a trial and error approach (see section 1.4.1 on page 33) with the tools used in this master's project. In a trial and error approach, you could for instance be using the primitive refactorings used in this project to refactor code, and only then make decisions based on the effect, possibly shown by traditional software metrics. The problem with the approach taken in this project, compared to a trial and error approach, is that using heuristics beforehand is much more complicated. But on the other hand, a trial and error approach would still need to face the challenges of producing code that does compile without errors. If using refactorings that could produce in-memory changes, a trial and error approach could be made more efficient.

Table 5.4: Statistics after batch refactoring the Eclipse JDT UI project with the *Extract and Move Method* refactoring.

Time used	
Total time	98m38s
Analysis time	14m41s (15%)
Change time	74m20s (75%)
Miscellaneous tasks	9m37s (10%)
Numbers of each type of entity analyzed	
Packages	110
Compilation units	2,097
Types	3,152
Methods	27,667
Text selections	591,500
Numbers for *Extract and Move Method* refactoring candidates	
Methods chosen as candidates	2,552
Methods NOT chosen as candidates	25,115
Candidate selections (multiple per method)	36,843
Extract and Move Method refactorings executed	
Fully executed	2,469
Not fully executed	83
Total attempts	2,552
Primitive refactorings executed *Extract Method* refactorings	
Performed	2,483
Not performed	69
Total attempts	2,552
Move Method refactorings	
Performed	2469
Not performed	14
Total attempts	2,483

Table 5.5: Results for analyzing the Eclipse JDT UI project, before and after the refactoring, with SonarQube and the IFI Refaktor Case Study quality profile. (Bold numbers are better.)

Number of issues for each rule	Before	After
Avoid too complex class	81	**79**
Classes should not be coupled to too many other classes (Single Responsibility Principle)	**1,098**	1,199
Control flow statements . . . should not be nested too deeply	1,375	**1,285**
Methods should not be too complex	1,518	**1,452**
Methods should not have too many lines	3,396	**3,291**
NPath Complexity	348	**329**
Too many methods	**454**	520
Total number of issues	8,270	**8,155**
Complexity		
Per function	3.6	**3.3**
Per class	**29.5**	30.4
Per file	**44.0**	45.3
Total complexity	**84,765**	87,257
Numbers of each type of entity analyzed		
Files	1,926	1,926
Classes	2,875	2,875
Functions	23,744	26,332
Accessors	1,296	1,019
Statements	162,768	165,145
Lines of code	320,941	329,112
Technical debt (in days)	**1,003.4**	1,032.7

Table 5.6: Results from the unit tests run for the Eclipse JDT UI project, before and after the refactoring.

AutomatedSuite	Before	After
Runs	2007/2007	2007/2007
Errors	4	565
Failures	3	5
AllAllRefactoringTests		
Runs	3815/3816	3815/3816
Errors	2	2257
Failures	3	0

5.6 Case 2: The `no.uio.ifi.refaktor` project

In this case we will see a form of the “dogfooding” methodology used, when refactoring our own **`no.uio.ifi.refaktor`** project with the *Extract and Move Method* refactoring.

In this case I will try to point out some differences from the first case, and how they impact the execution of the benchmark. The Refaktor project is 39 times smaller than the Eclipse JDT UI project, measured in lines of code. This will make things a bit more transparent. It will therefore be interesting to see if this case can shed light on any aspect of our refactoring that were lost in the larger first case.

The configuration for the experiment is specified in table 5.7.

Table 5.7: Configuration for Case 2.

Benchmark data	
Launch configuration	CaseStudyDogfooding.launch
Project	no.uio.ifi.refaktor.benchmark
Repository	gitolite@git.uio.no:ifi-stolz-refaktor
Commit	43c16c04520746edd75f8dc2a1935781d3d9de6c
Input data	
Project	no.uio.ifi.refaktor
Repository	gitolite@git.uio.no:ifi-stolz-refaktor
Commit	43c16c04520746edd75f8dc2a1935781d3d9de6c
Branch	master
Test configuration	no.uio.ifi.refaktor.tests/ExtractTest.launch

5.6.1 Statistics

The statistics gathered during the refactoring execution is presented in table 5.8 on page 102.

Differences

There are some differences between the two projects that make them a little difficult to compare by performance.

Different complexity. Although the JDT UI project is 39 times greater than the Refaktor project in terms of lines of code, it is only about 26 times its size measured in number of methods. This means that the average method size is smaller in the Refaktor project than in the JDT project. This is also reflected in the SonarQube report, where the cyclomatic complexity per method for the JDT project is 3.6, while the Refaktor project has a cyclomatic complexity of 2.1 per method.

Number of selections per method. The analysis for the JDT project processed 21 text selections per method in average. This number for the Refaktor project is only 8 selections per method analyzed. This is a direct consequence of smaller methods in the Refaktor project.

Different candidates to methods ratio. The differences in how the projects are factored are also reflected in the ratios for how many methods that are chosen as candidates compared to the total number of methods analyzed. For the JDT project, 9% of the methods were considered to be candidates, while for the Refaktor project, only 5% of the methods were chosen.

The average number of possible candidate selection. For the methods that are chosen as candidates, the average number of possible candidate selections differ quite much. For the JDT project, the number of possible candidate selections for these methods was 14.44 selections per method, while the candidate methods in the Refaktor project had only 3.91 candidate selections to choose from, in average.

Execution time

The differences in complexity, and the different candidate methods to total number of methods ratios, are shown in the distributions of the execution times. For the JDT project, 75% of the total time was used on the actual changes, while for the Refaktor project, this number was only 63%.

For the JDT project, the benchmark used on average 0.21 seconds per method in the project, while for the Refaktor project it used only 0.07 seconds per method. So the process used 3 times as much time per method for the JDT project than for the Refaktor project.

While the JDT project is 39 times larger than the Refaktor project measured in lines of code, the benchmark used about 79 times as long time on it than for the Refaktor project. Relatively, this is about twice as long.

Since the details of these execution times are not that relevant to this master's project, only their magnitude, I will leave them here.

Executed refactorings

For the composite *Extract and Move Method* refactoring performed in this case, 53 successful attempts out of 58 gives a success rate of 91.4%. This is 5.3 percentage points worse than for the first case.

5.6.2 SonarQube analysis

Results from the SonarQube analysis are shown in table 5.9 on page 103.

Not much is to be said about these results. The trends in complexity and coupling are the same as for the first case. We end up a little worse after the refactoring process than before.

5.6.3 Unit tests

The tests used for this case are the same that has been developed throughout this master's project.

The code that was refactored for this case suffered from some of the problems discovered in the first case. This means that the after-code for this case also contained compilation errors, but they were not as many. The code contained only 6 errors that made the code not compile.

All of the six errors originated from the same bug. The bug arises in a situation where a class instance creation is moved between packages, and the class for the instance is package-private. The *Move Method* refactoring does not detect that there will be a visibility problem, and neither does it promote the package-private class to be public.

Since the errors in the refactored Refaktor code were easy to fix manually, I corrected them and ran the unit tests as planned. The unit test results are shown in table 5.10 on page 104. Before the refactoring, all tests passed. All tests also passed after the refactoring, with the six error corrections. Since the corrections done are not of a kind that could make the behavior of the program change, it is likely that the refactorings done to the `no.uio.ifi.refaktor` project did not change its behavior. This is also supported by the informal experiment presented next.

5.6.4 An additional experiment

To complete the task of "eating my own dog food", I conducted an experiment where I used the refactored version of the `no.uio.ifi.refaktor` project, with the corrections, to again Refaktor "itself".

The experiment produced code containing the same six errors as after the previous experiment. I also compared the after-code from the two experiments with a diff-tool. The only differences found were different method names. This is expected, since the method names are randomly generated by the *Extract and Move Method* refactoring.

The outcome of this simple experiment makes me more confident that the *Extract and Move Method* refactoring made only behavior-preserving changes to the `no.uio.ifi.refaktor` project, apart from the compilation errors.

5.6.5 Conclusions

The differences in complexity between the Eclipse JDT UI project and the `no.uio.ifi.refaktor` project, clearly influenced the differences in their execution times. This is mostly because fewer of the methods were chosen to be refactored for the Refaktor project than for the JDT project. This makes it difficult to know if there are any severe performance penalties associated with refactoring on a large project compared to a small one.

The trends in the SonarQube analysis are the same for this case as for the previous one. This gives more confidence in these results.

By refactoring our own code and using it again to refactor our code, we showed that it is possible to write an automated composite refactoring that works for many cases. That it probably did not alter the behavior of a smaller project shows us nothing more than that though, and might just be a coincidence.

5.7 Threats to validity

Below there are listed some threats to the validity of the findings of the case studies presented in this chapter. These are characteristics of tools and methodologies that can make some conclusions less significant.

The measurement tool is not fine-grained enough. SonarQube, the measurement tool that is used to analyze source code both before and after changes are performed in the experiments, is not very fine-grained. Its approach of "rules" and "issues" makes entities either an issue, or not. This means that a class that has its coupling lowered by the *Extract and Move Method* refactoring, might still be an "issue" after the refactoring. SonarQube does not display such improvements. Although it is possible to find out to what degree a particular entity exceeds the limits of a rule, the trends of improvements, with respect to a particular rule, are not presented. This means that the property measured for a rule might improve overall, while the number of issues for the rule may increase at the same time, making SonarQube report it as degradation.

The choice of coupling metric. The CBO metric measures the number of couplings for a class, and in case of SonarQube, only outgoing dependencies are measured. For the search-based *Extract and Move Method* refactoring, it could make sense to measure the number of uses for each of these coupling relations. We could then observe how these numbers are influenced by the refactoring. Our choice of analysis tool limited our choices of software metrics that could be used for measuring coupling. By using a metric that measures coupling between classes, some results might be hidden from us that could otherwise put our refactoring in a better light, or further validate our conclusions.

The number of experiments. The number of projects that were subject to our experiments were limited. Only one large project was refactored, and although many different programmers have contributed to it, we could have had a better basis for drawing conclusions if more large projects were added to this list.

Each experiment was executed only once. This is not a big issue for our experiments, since most of the things we evaluated them by do not depend on the time used. But if we had executed each experiment several times, we would have had a better foundation for analyzing the performance of the search-based refactoring.

Table 5.8: Statistics after batch refactoring the **`no.uio.ifi.refaktor`** project with the *Extract and Move Method* refactoring.

Time used	
Total time	1m15s
Analysis time	0m18s (24%)
Change time	0m47s (63%)
Miscellaneous tasks	0m10s (14%)
Numbers of each type of entity analyzed	
Packages	33
Compilation units	154
Types	168
Methods	1,070
Text selections	8,609
Numbers for *Extract and Move Method* refactoring candidates	
Methods chosen as candidates	58
Methods NOT chosen as candidates	1,012
Candidate selections (multiple per method)	227
Extract and Move Method refactorings executed	
Fully executed	53
Not fully executed	5
Total attempts	58
Primitive refactorings executed *Extract Method* refactorings	
Performed	56
Not performed	2
Total attempts	58
Move Method refactorings	
Performed	53
Not performed	3
Total attempts	56

Table 5.9: Results for analyzing the `no.uio.ifi.refaktor` project, before and after the refactoring, with SonarQube and the IFI Refaktor Case Study quality profile. (Bold numbers are better.)

Number of issues for each rule	Before	After
Avoid too complex class	1	1
Classes should not be coupled to too many other classes (Single Responsibility Principle)	**29**	34
Control flow statements . . . should not be nested too deeply	24	**21**
Methods should not be too complex	17	**15**
Methods should not have too many lines	41	**40**
NPath Complexity	3	3
Too many methods	**13**	15
Total number of issues	**128**	129
Complexity		
Per function	2.1	2.1
Per class	**12.5**	12.9
Per file	**13.8**	14.2
Total complexity	**2,089**	2,148
Numbers of each type of entity analyzed		
Files	151	151
Classes	167	167
Functions	987	1,045
Accessors	35	30
Statements	3,355	3,416
Lines of code	8,238	8,460
Technical debt (in days)	**19.0**	20.7

Table 5.10: Results from the unit tests run for the **`no.uio.ifi.refaktor`** project, before and after the refactoring (with 6 corrections done to the refactored code).

	Before	After
Runs	148/148	148/148
Errors	0	0
Failures	0	0

Chapter 6

Conclusions and future work

This chapter will conclude this master's thesis. I will try to give justified answers to the research questions posed (see section 1.3.6 on page 30) and present some future work that could be done to take this project to the next level.

6.1 Conclusions

Some of the motivation for this thesis was to create a fully automated composite refactoring that could be used to make program source code better in terms of coupling between classes. Earlier, in section 1.3.5, it was shown that in an ideal situation, a composition of the *Extract Method* and *Move Method* refactorings reduces the coupling between two classes. The Eclipse JDT plugin implements both these refactorings, and also provides a framework for analyzing source code, so it was considered a suitable tool to build upon for our project.

The search-based *Extract and Move Method* refactoring was created by utilizing the analysis and refactoring support of Eclipse, and a small framework was built for executing large scale refactoring with it. The refactoring was set up to analyze and execute changes on the Eclipse JDT UI project. Statistics was gathered during this process and the resulting code was analyzed through SonarQube. The project's own unit tests were also performed to find out whether our refactoring introduces any behavior-altering changes in the code it refactors.

Answering the main research question. The first and greatest challenge was to find out if the *Extract and Move Method* refactoring could be automated, in all tasks ranging from analysis to executing changes. It is now confirmed that this can be done, since it has been implemented as a part of the work done for this project. It has also been shown that the refactoring can be used to refactor large code bases, through the case study done on the Eclipse JDT UI project.

Asking whether the existing Eclipse refactorings are well suited for this task is another question. The refactorings provided by the JDT UI project are clearly not meant to be combined in any way. The preconditions for one refactoring are not always easily retrievable after the execution of another. Also, the refactorings are all assuming that the code they are going to refactor is textualized. This means that the source code must be parsed between the executions of each refactoring. Another problem with this dependency on textual changes, is that you cannot make a composition of two refactorings appear as one change if the two refactorings' changes overlap. This will make the undo-history of the composite refactoring show two changes instead of one, and is not nice for usability if the refactoring would be used as an on-demand refactoring in an IDE.

Apart from the problems with implementing the actual refactoring, the analysis framework is quite nicely solved in Eclipse. The AST generated when parsing source code, supports using visitors to traverse it, and this works without problems.

Is the refactoring efficient enough? Since we have concluded that the search-based *Extract and Move Method* refactoring is not suitable for on-demand large-scale refactoring, but may be put to better use as a kind of analysis tool, superb performance is not crucial. In section 5.5.1 on page 86 we conclude that the refactoring performs well enough for this purpose. If performed on demand for a single method, the performance of the *Extract and Move Method* refactoring is not an issue.

What about breaking the source code? The case studies show that our safety measures, which rely on the precondition checking of the existing primitive refactorings, are not good enough in practice. If we were going to assure that code we refactor compiles, we would need to consider all possible situations where the refactoring could fail, and search for them in our analysis. It is an open question if this is even feasible. Our analysis is incomplete, and so are the analyses for the *Extract Method* and *Move Method* refactorings.

Our refactoring does not take any precautions to preserve behavior. A few running and failing unit test for the JDT UI project after the refactoring indicate that our refactoring causes some changes to the way a program behaves.

Is the quality of source code improved? For coupling, there is no evidence that the refactoring improves the quality of source code. Shall we believe the SonarQube analysis from the case studies, our refactoring makes classes more coupled after the refactoring than before, in the general case. Examples exist where the *Extract and Move Method* refactoring improves coupling. The problem is that it introduces too many dependencies overall (see section 5.5.2 on page 89). The essence is that our analysis and heuristics for finding the best candidates for the refactoring are not adequate.

Is the refactoring useful? In its present state, the refactoring cannot be said to be very useful. It generates too many compilation errors for it to fall into that category. On the other hand, if the problems with the search-based

Extract and Move Method refactoring were to be solved, it could be put to use in some situations.

If the refactoring was perfected, it could of course be used as a regular on-demand automated refactoring on a per method base (or per class, package or project).

As it is now, the refactoring is not well suited for performing unattended refactoring. But if we could find a way to filter out the changes that create compilation errors, we could use the refactoring to look for improvement points in software projects. This process could for instance be scheduled to run at regular intervals, possibly after a nightly build or the like. Then the results could be made available, and an administrator could be set to review them and choose whether or not they should be performed.

6.2 Future work

An important part that is missing for making the search-based *Extract and Move Method* refactoring more usable, is to complete the pre-refactoring analysis of the source code, to make sure that the refactoring does not introduce compilation errors when it is performed.

The first point of making the static analysis complete, brings up the next question: Is it feasible to complete such an analysis? And can this feasibility be proven, or disproved?

Another shortcoming of this project is that we have no strategy for assuring safety when refactoring, so a program may end up behaving differently after a refactoring than it behaved before. One approach toward safer refactorings is mentioned in section 1.4.2 on page 33, and includes generating tests for the refactored code. Another approach that can be considered for making refactorings safer is part of the original thesis proposal for this thesis, which diverged somewhat from the original proposal. The proposal is about detecting behavioral changes during refactoring, and the work done in this thesis can be used as a basis if one would like to engage in that proposal. The proposed solution for exposing behavioral changes, is to insert assertions into source code when refactoring it. For the example in listing 3 on page 25, which is the result of a refactoring, it is suggested that we insert an assert statement between lines 9 and 10. In the example, the assert statement would be

```
assert c.x == this;
```

which would discover the aliasing problems of the example.

The final important improvement that I would suggest making to this project, is to refine the heuristics that are used to find suitable refactoring candidates. This effort should in particular be directed toward making the heuristics choose candidates that do not introduce new dependencies to their destination classes.

Appendix A

Eclipse bugs submitted

A.1 Eclipse bug 420726: Code is broken when moving a method that is assigning to the parameter that is also the move destination

This bug was found when analyzing what kinds of names that were to be considered as *unfixes* (see section 2.7 on page 46).

The bug. The bug emerges when trying to move a method from one class to another, and when the target for the move (must be a variable, local or field) is both a parameter variable and also is assigned to within the method body. Eclipse allows this to happen, although it is the sure path to a compilation error. This is because we would then have an assignment to a `this` expression, which is not allowed in Java. The submitted bug report can be found on https://bugs.eclipse.org/bugs/show_bug.cgi?id=420726.

The solution. The solution to this problem is to add all simple names that are assigned to in a method body to the set of unfixes.

A.2 Eclipse bug 429416: IAE when moving method from anonymous class

I discovered this bug during a batch change on the `org.eclipse.jdt.ui` project.

The bug. This bug surfaces when trying to use the *Move Method* refactoring to move a method from an anonymous class to another class. This happens both for my simulation as well as in Eclipse, through the user interface. It only occurs when Eclipse analyzes the program and finds it necessary to pass an instance of the originating class as a parameter to the moved method. I.e. it wants to pass a `this` expression. The execution ends in an `IllegalArgumentException`[1] in

[1] `java.lang.IllegalArgumentException`

`SimpleName`[1] and its `setIdentifier(String)` method. The simple name is attempted created in the method `createInlinedMethodInvocation`[2] so the `MoveInstanceMethodProcessor` was early a clear suspect.

The `createInlinedMethodInvocation` is the method that creates a method invocation where the previous invocation to the method that was moved was located. From its code it can be read that when a `this` expression is going to be passed in to the invocation, it shall be qualified with the name of the original method's declaring class, if the declaring class is either an anonymous class or a member class. The problem with this, is that an anonymous class does not have a name, hence the term *anonymous* class! Therefore, when its name, an empty string, is passed into `newSimpleName`[3] it all ends in an `IllegalArgumentException`. The submitted bug report can be found on https://bugs.eclipse.org/bugs/show_bug.cgi?id=429416.

How I solved the problem. Since the `MoveInstanceMethodProcessor` is instantiated in the `MoveMethodRefactoringExecutor`[4], and only need to be a `MoveProcessor`[5], I was able to copy the code for the original move processor and modify it so that it works better for me. It is now called `ModifiedMoveInstanceMethodProcessor`[6]. The only modification done (in addition to some imports and suppression of warnings), is in the `createInlinedMethodInvocation`. When the declaring class of the method to move is anonymous, the `this` expression in the parameter list is not qualified with the declaring class' (empty) name.

A.3 Eclipse bug 429954: Extracting statement with reference to local type breaks code

The bug was discovered when doing some changes to the way unfixes is computed.

The bug. The problem is that Eclipse is allowing selections that references variables of local types to be extracted. When this happens the code is broken, since the extracted method must take a parameter of a local type that is not in the methods scope. The problem is illustrated in listing 10 on page 47, but there in another setting. The submitted bug report can be found on https://bugs.eclipse.org/bugs/show_bug.cgi?id=429954.

Actions taken. There are no actions directly springing out of this bug, since the Extract Method refactoring cannot be meant to be this way. This is handled on the analysis stage of our *Extract and Move Method* refactoring.

[1] `org.eclipse.jdt.core.dom.SimpleName`

[2] `org.eclipse.jdt.internal.corext.refactoring.structure.MoveInstanceMethodProcessor#createInlinedMethodInvocation()`

[3] `org.eclipse.jdt.core.dom.AST#newSimpleName()`

[4] `no.uio.ifi.refaktor.change.executors.MoveMethodRefactoringExecutor`

[5] `org.eclipse.ltk.core.refactoring.participants.MoveProcessor`

[6] `no.uio.ifi.refaktor.change.processors.ModifiedMoveInstanceMethodProcessor`

So names representing variables of local types are considered unfixes (see section 2.7 on page 46).

Appendix B

Continuous integration

The continuous integration server Jenkins has been set up for the project[1]. It is used as a way to run tests and perform code coverage analysis.

To be able to build the Eclipse plugins and run tests for them with Jenkins, the component assembly project Buckminster is used, through its plugin for Jenkins. Buckminster provides for a way to specify the resources needed for building a project and where and how to find them. Buckminster also handles the setup of a target environment to run the tests in. All this is needed because the code to build depends on an Eclipse installation with various plugins.

Problems with AspectJ

The Buckminster build worked fine until introducing AspectJ into the project. When building projects using AspectJ, there are some additional steps that need to be performed. First of all, the aspects themselves must be compiled. Then the aspects need to be woven with the classes they affect. This demands a process that does multiple passes over the source code.

When using AspectJ with Eclipse, the specialized compilation and the weaving can be handled by the AspectJ Development Tools. This works all fine, but it complicates things when trying to build a project depending on Eclipse plugins outside of Eclipse. There is supposed to be a way to specify a compiler adapter for javac, together with the file extensions for the file types it shall operate. The AspectJ compiler adapter is called `Ajc11CompilerAdapter`[2], and it works with files that has the extensions `*.java` and `*.aj`. I tried to setup this in the build properties file for the project containing the aspects, but to no avail. The project containing the aspects does not seem to be built at all, and the projects that depend on it complain that they cannot find certain classes.

I then managed to write an Ant build file that utilizes the AspectJ compiler adapter, for the `no.uio.ifi.refaktor` plugin. The problem was then that it could no longer take advantage of the environment set up by Buckminster. The solution to this particular problem was of a "hacky"

[1]A work mostly done by the supervisor.

[2]`org.aspectj.tools.ant.taskdefs.Ajc11CompilerAdapter`

nature. It involves exporting the plugin dependencies for the project to an Ant build file, and copy the exported path into the existing build script. But then the Ant script needs to know where the local Eclipse installation is located. This is no problem when building on a local machine, but to utilize the setup done by Buckminster is a problem still unsolved. To get the classpath for the build setup correctly, and here comes the most “hacky” part of the solution, the Ant script has a target for copying the classpath elements into a directory relative to the project directory and checking it into Git. When no `ECLIPSE_HOME` property is set while running Ant, the script uses the copied plugins instead of the ones provided by the Eclipse installation when building the project. This obviously creates some problems with maintaining the list of dependencies in the Ant file, as well as remembering to copy the plugins every time the list of dependencies changes.

The Ant script described above is run by Jenkins before the Buckminster setup and build. When setup like this, the Buckminster build succeeds for the projects not using AspectJ, and the tests are run as normal. This is all good, but it feels a little scary, since the reason for Buckminster not working with AspectJ is still unknown.

The problems with building with AspectJ on the Jenkins server lasted for a while, before they were solved. This is reflected in the “Test Result Trend” and “Code Coverage Trend” reported by Jenkins.

Glossary

design pattern A design pattern is a named abstraction that is meant to solve a general design problem. It describes the key aspects of a common problem and identifies its participators and how they collaborate. 15

enclosing class An enclosing class is the class that surrounds any specific piece of code that is written in the inner scope of this class. 41

Extract Class The *Extract Class* refactoring works by creating a class, for then to move members from another class to that class and access them from the old class via a reference to the new class. 20

memento pattern The memento pattern is a software design pattern that is used to capture an object's internal state so that it can be restored to this state later [Gam+95]. 81

profiler A profiler is a program for analyzing performance within an application. It is used to analyze memory consumption, processing time and frequency of procedure calls and such. 80

profiling is to run a computer program through a profiler/with a profiler attached. 22, 80

software obfuscation makes source code harder to read and analyze, while preserving its semantics. 14

xUnit framework An xUnit framework is a framework for writing unit tests for a computer program. It follows the patterns known from the JUnit framework for Java [Fow]. 26

References

[11] *JAVA EE Productivity Report 2011.* Survey. 2011. URL: http://zeroturnaround.com/wp-content/uploads/2010/11/Java_EE_Productivity_Report_2011_finalv2.pdf.

[Ban+93] Rajiv D. Banker et al. "Software Complexity and Maintenance Costs." In: *Commun. ACM* 36.11 (Nov. 1993), 81–94. (Visited on 04/29/2014).

[Bro04] Leo Brodie. *Thinking Forth.* 3rd ed. 2004. URL: http://thinking-forth.sourceforge.net/.

[CK94] S.R. Chidamber and C.F. Kemerer. "A Metrics Suite for Object Oriented Design." In: *IEEE Transactions on Software Engineering* 20.6 (June 1994), pp. 476–493.

[Dem02] Serge Demeyer. "Maintainability Versus Performance: What's the Effect of Introducing Polymorphism?" In: *ICSE'2003* (2002).

[Fow] Martin Fowler. *Xunit.* URL: http://www.martinfowler.com/bliki/Xunit.html (visited on 03/27/2014).

[Fow01] Martin Fowler. *Crossing Refactoring's Rubicon.* 2001. URL: http://martinfowler.com/articles/refactoringRubicon.html (visited on 02/09/2014).

[Fow03] Martin Fowler. *EtymologyOfRefactoring.* Sept. 10, 2003. URL: http://martinfowler.com/bliki/EtymologyOfRefactoring.html (visited on 03/20/2014).

[Fow04] Martin Fowler. *Is Design Dead?* 2004. URL: http://martinfowler.com/articles/designDead.html (visited on 04/09/2014).

[Fow99] Martin Fowler. *Refactoring: improving the design of existing code.* Reading, MA: Addison-Wesley, 1999.

[Gam+95] Erich Gamma et al. *Design patterns: elements of reusable object-oriented software.* Reading, MA: Addison-Wesley, 1995.

[Har06] W. Harrison. "Eating Your Own Dog Food." In: *IEEE Software* 23.3 (May 2006), pp. 5–7.

[Ker05] Joshua Kerievsky. *Refactoring to patterns.* Boston: Addison-Wesley, 2005.

[Lou97] Kenneth C Louden. *Compiler construction: principles and practice.* Boston: PWS Pub. Co., 1997.

[MC09] Robert C Martin and James O Coplien. *Clean code: a handbook of agile software craftsmanship*. Upper Saddle River, NJ [etc.]: Prentice Hall, 2009.

[McC76] T.J. McCabe. “A Complexity Measure.” In: *IEEE Transactions on Software Engineering* SE-2.4 (Dec. 1976), pp. 308–320.

[Mey88] Bertrand Meyer. *Object-oriented software construction*. Prentice-Hall, 1988.

[Mil56] George A. Miller. “The magical number seven, plus or minus two: some limits on our capacity for processing information.” In: *Psychological Review* 63.2 (1956), pp. 81–97.

[Nic06] Ethan Nicholas. *Understanding Weak References*. Java.net. May 4, 2006. URL: https://weblogs.java.net/blog/2006/05/04/understanding-weak-references (visited on 03/20/2014).

[OC08] Mark O’Keeffe and Mel Ó Cinnéide. “Search-based Refactoring: An Empirical Study.” In: *J. Softw. Maint. Evol.* 20.5 (Sept. 2008), 345–364.

[Opd92] William F. Opdyke. “Refactoring Object-oriented Frameworks.” UMI Order No. GAX93-05645. Champaign, IL, USA: University of Illinois at Urbana-Champaign, 1992.

[RBJ97] Don Roberts, John Brant, and Ralph Johnson. “A Refactoring Tool for Smalltalk.” In: *Theor. Pract. Object Syst.* 3.4 (Oct. 1997), 253–263.

[Sha10] R. Shatnawi. “A Quantitative Investigation of the Acceptable Risk Levels of Object-Oriented Metrics in Open-Source Systems.” In: *IEEE Transactions on Software Engineering* 36.2 (Mar. 2010), pp. 216–225.

[Sho04] Jim Shore. “Continuous Design.” In: *IEEE Softw.* 21.1 (2004), 20–22.

[Soa+10] G. Soares et al. “Making Program Refactoring Safer.” In: *IEEE Software* 27.4 (Aug. 2010), pp. 52 –57.

[Vak+12] Mohsen Vakilian et al. *A Compositional Paradigm of Automating Refactorings*. May 2012. URL: https://www.ideals.illinois.edu/bitstream/handle/2142/30851/VakilianETAL2012Compositional.pdf?sequence=4.

[VJ12] Mohsen Vakilian and Ralph Johnson. *Composite Refactorings: The Next Refactoring Rubicons*. University of Illinois at Urbana-Champaign, 2012. URL: https://www.ideals.illinois.edu/bitstream/handle/2142/35678/2012-WRT.pdf?sequence=2.

[Vog12] Lars Vogel. *Eclipse JDT - Abstract Syntax Tree (AST) and the Java Model - Tutorial*. vogella. Aug. 8, 2012. URL: http://www.vogella.com/tutorials/EclipseJDT/article.html (visited on 04/20/2014).