Automated Composition of Refactorings

Composing the Extract and Move Method refactorings in Eclipse

Erlend Kristiansen
Master's Thesis Spring 2014

Abstract

Remove all todos (including list) before delivery/printing!!!
Can be done by removing "draft" from documentclass.

Write abstract

Contents

List of Figures

List of Tables

x

Preface

The discussions in this report must be seen in the context of object oriented programming languages, and Java in particular, since that is the language in which most of the examples will be given. All though the techniques discussed may be applicable to languages from other paradigms, they will not be the subject of this report.

Chapter 1

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.

1.1 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*, that 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.

---

[1]The structure observable by the programmer.

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 1.9 on page 10)? 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 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 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[1] community and their infamous Refactoring Browser[2] 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[1] community, and the word "refactoring" is mentioned in a book by Leo Brodie, called *Thinking Forth* [Bro04], first published in 1984[3]. The exact word is only printed one place [Bro04, p. 232], but the term *factoring* is prominent in the book, that 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

---

[1]Programming language

[2]http://st-www.cs.illinois.edu/users/brant/Refactory/RefactoringBrowser.html

[3]*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 licence in 2004. The edition cited here is the 2004 edition, but the content should essentially be as in 1984.

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.

1.3 Motivation – Why people refactor

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
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 `Proof?`
giving good names to the new ones clarifies the algorithms and enhances the
understandability of the program (see section 1.4 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 1.9 on page 10). 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

---

[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.

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 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.

1.4   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 suggest 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.

1.5 Notable contributions to the refactoring literature

Update with more contributions

5

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 1.7 on the next page).

1.6 Tool support (for Java)

This section will briefly compare the refactoring support of the three IDEs Eclipse[1], IntelliJ IDEA[2] and NetBeans[3]. 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, 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 1.11 on page 11). 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

---

[1] http://www.eclipse.org/

[2] The IDE under comparison is the Community Edition, http://www.jetbrains.com/idea/

[3] https://netbeans.org/

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 on the current page.

```
public class C {                    public class X {
    private X x;                        ...
    ...                                 public void f(C c) {
    public void f() {                       c.x.m();
        x.m();                              c.x.n();
        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 code in the example from section 1.11 on page 11.) 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.

Visual Studio (C++/C#), Smalltalk refactoring browser?, second refactoring rubicon?

1.7 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]

7

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.[1]

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

---

[1]See http://c2.com/cgi/wiki?OpenClosedPrinciple or https://en.wikipedia.org/wiki/Open/closed_principle

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 1.12 on page 12). 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.

1.8 The impact on software quality

1.8.1 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.)

1.8.2 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.)

---

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

> 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.

1.9   Composite refactorings

motivation, examples, manual vs automated?, what about refactoring in a very large code base?

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 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 fig. 1.1 on the facing page.

1.10   Manual vs. automated refactorings

Refactoring is something every programmer does, even if she does not known 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
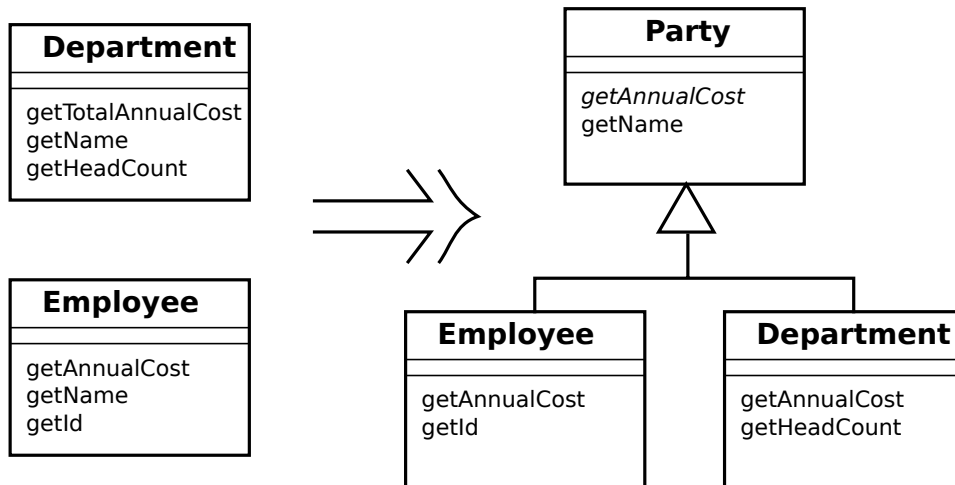
Figure 1.1: The Extract Superclass refactoring, with united interfaces.

to what construct in the program. For even trying to perform one of these complex task manually, one would have to be very confident on the existing test suite (see section 1.12 on the next page).

1.11 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, that 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 is shown in listing 2 on the next page. Note that the method `m(C c)` of class `X` assigns to the field `x` of the argument `c` that has type `C`.

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 on the following page.

Before the refactoring, the methods `m` and `n` of class `X` are called on

---

[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 1.6 on page 6).

```
1  // Refactoring target        // Method destination
2  public class C {             public class X {
3    public X x = new X();         public void m(C c) {
4                                     c.x = new X();
5    public void f() {               // If m is called from
6      x.m(this);                    // c, then c.x no longer
7      // Not the same x            // equals 'this'
8      x.n();                     }
9    }                           public void n() {}
10 }                           }
```

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

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.

```
1  public class C {             1  public class X {
2      public X x = new X();     2      public void m(C c) {
3                                3          c.x = new X();
4      public void f() {         4      }
5          x.f(this);            5      public void n() {}
6      }                         6      // Extracted and
7  }                             7      // moved method
                                 8      public void f(C c) {
                                 9          m(c);
                                 10         n();
                                 11     }
                                 12 }
```

Listing 3: The result of the composed refactoring.

The bug introduced in the previous example is of such a nature[1] 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.

1.12 Refactoring and the importance of testing

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

---

[1]Caused by aliasing. See https://en.wikipedia.org/wiki/Aliasing_(computing)

When refactoring, there are roughly three classes of errors that can be made. The first class of errors are the ones that make 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 are the *runtime* errors. Although they 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 kind 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 kind of errors does 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 are such a great match.

1.12.1   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 1.11 on page 11. 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.

Write?

Chapter 2

The Project

> Moved from introduction to here. Rewrite and make problem statement from it:

The aim of this master project will be to investigate the relationship between a composite refactoring composed of the *Extract Method* and *Move Method* refactorings, and its impact on one or more software metrics.

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[1], that is an extension for MS Visual Studio[2]. In CodeRush it is called *Extract Method to Type*[3], but I choose to call it *Extract and Move Method*, since I feel it better communicates which primitive refactorings it is composed of.

For the metrics, I will at least measure the *Coupling between object classes* (CBO) metric that is described by Chidamber and Kemerer in their article *A Metrics Suite for Object Oriented Design* [CK94].

The project will then consist in implementing the *Extract and Move Method* refactoring, as well as executing it over a larger code base. Then the effect of the change must be measured by calculating the chosen software metrics both before and after the execution. 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.

2.1   The problem statement

> write/move

---

[1] https://help.devexpress.com/#CodeRush/CustomDocument3519
[2] http://www.visualstudio.com/
[3] https://help.devexpress.com/#CodeRush/CustomDocument6710

2.2 Choosing the target language

Choosing which programming language the code that shall 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[1] is the dominating language when it comes to example code in the literature of refactoring, and is thus a natural choice. Java is perhaps, currently the most influential programming language in the world, with its Java Virtual Machine that runs on all of the most popular architectures and also supports dozens of other programming languages[2], 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.

2.3 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, that 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*[3]. These are programs that is 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 is the Eclipse IDE, with the Java development tools (JDT), the IntelliJ IDEA Community Edition and the NetBeans IDE (see section 1.6 on page 6). 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 offer 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 IDEs stand out as a favorite, and that is the Eclipse IDE. This is the most

---

[1] https://www.java.com/

[2] They compile to java bytecode.

[3] *Integrated Development Environment*

popular [11] among them and seems to be de facto standard IDE for Java development regardless of platform.

2.4 Organizing the project

All the parts of this master project is under version control with Git[1].

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 projects/packages.

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

2.5 Continuous integration

The continuous integration server Jenkins[2] has been set up for the project[3]. 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[4] 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.

2.5.1 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 needs to be performed. First of all, the aspects themselves must be compiled. Then the aspects needs 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[5]. 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`[6], and it works with files that has the extensions `*.java` and `*.aj`. I tried to setup this in the build properties file for the

---

[1] http://git-scm.com/

[2] http://jenkins-ci.org/

[3] A work mostly done by the supervisor.

[4] http://www.eclipse.org/buckminster/

[5] https://www.eclipse.org/ajdt/

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

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 depends on it complains that they cannot find certain classes.

I then managed to write an Ant[1] 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" 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 change.

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.

---

[1] https://ant.apache.org/

Chapter 3

Refactorings in Eclipse JDT: Design, Shortcomings and Wishful Thinking

This chapter will deal with some of the design behind 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. The chapter will be concluded with a section telling some of the ways the implementation of refactorings in the JDT could have worked to facilitate composition of refactorings.

3.1 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 refactorings.

> What about the language specific part?

3.1.1 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.

---

[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`

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 have to end up creating an instance of one of its subclasses. The main responsibilities of subclasses of `Refactoring` is to implement template methods for condition checking (`checkInitialConditions`[1] and `checkFinalConditions`[2]), in addition to the `createChange`[3] 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[4]. It then delegates to its given `RefactoringProcessor`[5] for condition checking and change creation. Participating in a refactoring can be useful in cases where the changes done to programming source code affects 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 is responsible for performing the actual workspace transformations in a refactoring. The main responsibilities for its subclasses is to implement the `perform`[6] and `isValid`[7] 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 that can be executed to reverse the effect of the transformation done by its originating change object.

Executing a Refactoring

The life cycle of a refactoring generally follows two steps after creation: condition checking and change creation. By letting the refactoring object be handled by a `CheckConditionsOperation`[8] that in turn is handled by a `CreateChangeOperation`[9], 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`[10]. If also an undo manager[11] is set for the `PerformChangeOperation`, the undo change is added into the undo history.

---

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

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

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

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

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

[6] `org.eclipse.ltk.core.refactoring.Change#perform()`

[7] `org.eclipse.ltk.core.refactoring.Change#isValid()`

[8] `org.eclipse.ltk.core.refactoring.CheckConditionsOperation`

[9] `org.eclipse.ltk.core.refactoring.CreateChangeOperation`

[10] `org.eclipse.ltk.core.refactoring.PerformChangeOperation`

[11] `org.eclipse.ltk.core.refactoring.IUndoManager`

3.2 Shortcomings

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

refine

I will begin at the end and work my way toward the composition part of this section.

3.2.1 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.

3.2.2 Composite Refactorings Will Not Appear as Atomic Actions

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 composed refactoring that is to be performed as a sequence of 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 is 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 4.2.6 on page 31).

3.3 Wishful Thinking

???

Chapter 4

Composite Refactorings in Eclipse

4.1 A Simple Ad Hoc Model

As pointed out in chapter 3 on page 19, 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.

4.1.1 A typical `RefaktorChanger`

The typical refaktor changer class has two responsibilities, checking
preconditions and executing the requested 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 the job.

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

4.2 The Extract and Move Method Refactoring

4.2.1 The Building Blocks

This is a composite refactoring, and hence is built up using several primitive
refactorings. These basic building blocks are, as its name implies, the *Extract*

---

[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`

Method refactoring [Fow99] and the *Move Method* refactoring [Fow99]. In Eclipse, the implementations of these refactorings are found in the classes `ExtractMethodRefactoring`[1] and `MoveInstanceMethodProcessor`[2], where the last class is designed to be used together with the processor-based `MoveRefactoring`[3].

The ExtractMethodRefactoring Class

This class is quite simple in its use. The only parameters it requires for construction is a compilation unit[4], 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*, the processor requires a little more advanced input than the class for the *Extract Method*. For construction it requires a method handle[5] for the method that is to be moved. Then the target for the move have to be supplied as the variable binding from a chosen variable declaration. In addition to this, one have to set some parameters regarding setters/getters, as well as delegation.

To make a working refactoring from the processor, one have to create a `MoveRefactoring` with it.

4.2.2 The ExtractAndMoveMethodChanger

The `ExtractAndMoveMethodChanger`[6] class is a subclass of the class `RefaktorChanger`[7]. It is responsible for analyzing and finding the best target for, and also executing, a composition of the *Extract Method* and *Move Method* refactorings. 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, or else a method name will be generated. The selection has to be of the type `CompilationUnitTextSelection`[8]. This class is a custom extension to `TextSelection`[9], that in addition to the basic offset, length and similar methods, also carry an instance of the underlying compilation unit handle for the selection.

---

[1]`org.eclipse.jdt.internal.corext.refactoring.code.ExtractMethodRefactoring`
[2]`org.eclipse.jdt.internal.corext.refactoring.structure.MoveInstanceMethodProcessor`
[3]`org.eclipse.ltk.core.refactoring.participants.MoveRefactoring`
[4]`org.eclipse.jdt.core.ICompilationUnit`
[5]`org.eclipse.jdt.core.IMethod`
[6]`no.uio.ifi.refaktor.changers.ExtractAndMoveMethodChanger`
[7]`no.uio.ifi.refaktor.changers.RefaktorChanger`
[8]`no.uio.ifi.refaktor.utils.CompilationUnitTextSelection`
[9]`org.eclipse.jface.text.TextSelection`

The `ExtractAndMoveMethodAnalyzer`

The analysis and precondition checking is done by the **ExtractAnd-MoveMethodAnalyzer**[1]. First is check whether the 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 is performed by a range of checkers (see section 5.5 on page 41). 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**[2] that collects all the possible candidate targets for the refactoring. All the non-candidates is found by an **UnfixesCollector**[3] that collects all the targets that will give some kind of error if used. (For details about the property collectors, see section 5.4 on page 39.) All prefixes (and unfixes) are represented by a **Prefix**[4], and they are collected into sets of prefixes. The safe prefixes is found by subtracting from the set of candidate prefixes the prefixes that is 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 length, the longest of them is chosen.

> Clean up sections/subsections.

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 **MoveMethod-RefactoringExecutor** with the resources needed after executing the extract method refactoring.

---

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

[2] `no.uio.ifi.refaktor.analyze.collectors.PrefixesCollector`

[3] `no.uio.ifi.refaktor.analyze.collectors.UnfixesCollector`

[4] `no.uio.ifi.refaktor.extractors.Prefix`

[5] `no.uio.ifi.refaktor.change.executors.ExtractAndMoveMethodExecutor`

[6] `no.uio.ifi.refaktor.change.executors.ExtractMethodRefactoringExecutor`

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

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 `MoveInstanceMethodProcessor` that is used.

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 find the method handle, is the textual representation 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.

4.2.3 The SearchBasedExtractAndMoveMethodChanger

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**[2] 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 4.2.2 on page 25).

The SearchBasedExtractAndMoveMethodAnalyzer

This analyzer is responsible for analyzing all the possible text selections of a method and then choose the best result out of the analysis results that is, 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 5.2.1 on page 36). Then there is a statement lists creator that creates statements 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 levels of statements in the method. The statement lists creator is implemented as an AST visitor (see section 5.3 on page 37). 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.

There is an example of how the statement lists creator would generate lists for a simple method in listing 4 on the following page.

The text selections generator generates text selections for each list of statements from the statement lists creator. Conceptually, the generator generates a text selection for every possible ordered combination [make clearer] of statements in a list. For a list of statements, the boundary statements

---

[1] `no.uio.ifi.refaktor.change.changers.SearchBasedExtractAndMoveMethodChanger`
[2] `no.uio.ifi.refaktor.analyze.analyzers.SearchBasedExtractAndMoveMethodAnalyzer`

```
void method() {
    if (bool)
        b.bar();

    switch (val) {
        case 1:
            b.foo();
            c.foo();
        default:
            c.foo();
    }
}
```

Listing 4: Example of how the statement lists creator would group a simple method into lists of statements. Each highlighted rectangle represents a list.

span out a text selection. This means that there are many different lists that could span out the same selection.

In practice, the text selections are calculated by only one traversal of 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.

```
statement one;
statement two;
...
statement k;
```

Listing 5: Example of how the text selections generator would generate text selections based on a lists of statements. Each highlighted rectangle represents a text selection.

fix listing 5 on the current page?

Finding the candidate for the refactoring is done by analyzing all the generated text selection with the `ExtractAndMoveMethodAnalyzer` (see section 4.2.2 on page 25). If the analyzer generates a useful result, an `ExtractAndMoveMethodCandidate` is created from it, that 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`.

> Write about the ExtractAndMoveMethodCandidateComparator/FavorNoUnfixesCandidateComparator

The complexity of how many text selections that needs to be analyzed for a total of n statements is bounded by $O(n^2)$.

Theorem. The number of text selections that need to be analyzed for each list 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. For the second, you get one selection for the statement itself, 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: $\frac{k(k+1)}{2} + k + 1 = \frac{k(k+1)+2k+2}{2} = \frac{k(k+1)+2(k+1)}{2} = \frac{(k+1)(k+2)}{2} = \frac{(k+1)((k+1)+1)}{2} = \sum_{i=1}^{k+1} i$ ☐

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. Let $l, \cdots, m, (k - l - \cdots - m)$ be the lengths of the lists of statements in the body, with $l + \cdots + m < k \Rightarrow l, \cdots, m < k$.

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

$$\frac{(k-l-\cdots-m)((k-l-\cdots-m)+1)}{2} + \frac{l(l+1)}{2} + \cdots + \frac{m(m+1)}{2} =$$

$$\frac{k^2 - 2kl - \cdots - 2km + l^2 + \cdots + m^2 + k - l - \cdots - m}{2} + \frac{l^2 + l}{2} + \cdots + \frac{m^2 + m}{2} =$$

$$\frac{k^2 + k + 2l^2 - 2kl + \cdots + 2m^2 - 2km}{2}$$

It then remains to show that this inequality holds:

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

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

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

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

$\square$

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

4.2.4 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.

4.2.5 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.

---

[1] `java.util.HashMap`
[2] `java.util.Set`

There are two ways 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.

4.2.6 Hacking the Refactoring Undo History

Where to put this section?

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[1] in the history of the refactorings.

My first impulse was to remove the, in this case, last two undo changes from the undo manager[2] for the Eclipse refactorings, and then add them to a composite change[3] 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[4] 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[5]. 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[6]. 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.

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.

---

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

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

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

[4] `org.eclipse.ltk.core.refactoring.IUndoManager#addUndo()`

[5] `org.eclipse.ltk.internal.core.refactoring.UndoManager2`

[6] `org.eclipse.core.commands.operations.TriggeredOperations`

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 change done by its originating refactoring. And in addition, the undo actions has to be performed separate 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 it is to complex to be easily manipulated.

Chapter 5

Analyzing Source Code in Eclipse

5.1 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 is 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 handles with descriptions is listed in table 5.1 on the current page.

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.

Table 5.1: The elements of the Java Model. Taken from http://www.vogella.com/tutorials/EclipseJDT/article.html

The hierarchy of the Java Model is shown in fig. 5.1 on this page.

```
IJavaProject
 └─IPackageFragmentRoot
    └─IPackageFragment
       └─ICompilationUnit
          └─IType
             ├─{ IType }*
             │  └─...
             ├─{ IField }*
             ├─IMethod
             │  └─{ IType }*
             │     └─...
             └─{ IMethod }*
          └─{ IType }*
       └─{ ICompilationUnit }*
    └─{ IPackageFragment }*
 └─{ IPackageFragmentRoot }*
```
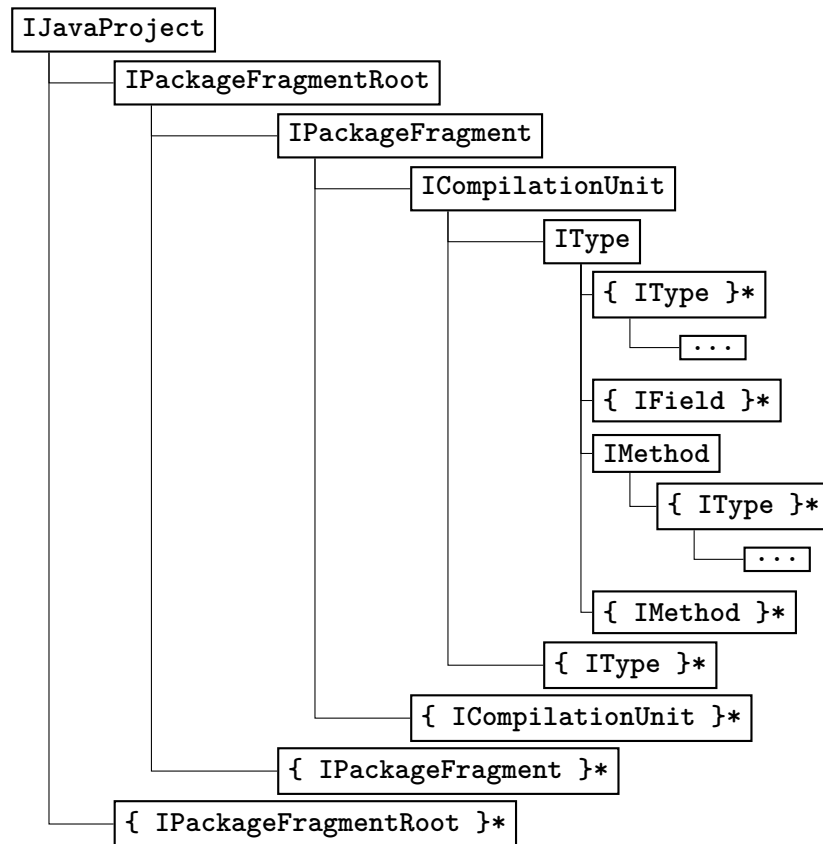
Figure 5.1: The Java model of Eclipse. "**{ SomeElement }***" means **SomeElement** zero or more times. For recursive structures, "**...**" is used.

5.2   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 no different from how source manipulation programs would do it, except for some properties of code that is analyzed in the parser, and that they may be differing in what kinds of properties they analyze. Thus the process of translation source code into a structure that is suitable for analyzing, can be seen as a kind of interrupted compilation process (see fig. 5.2 on the next page).

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
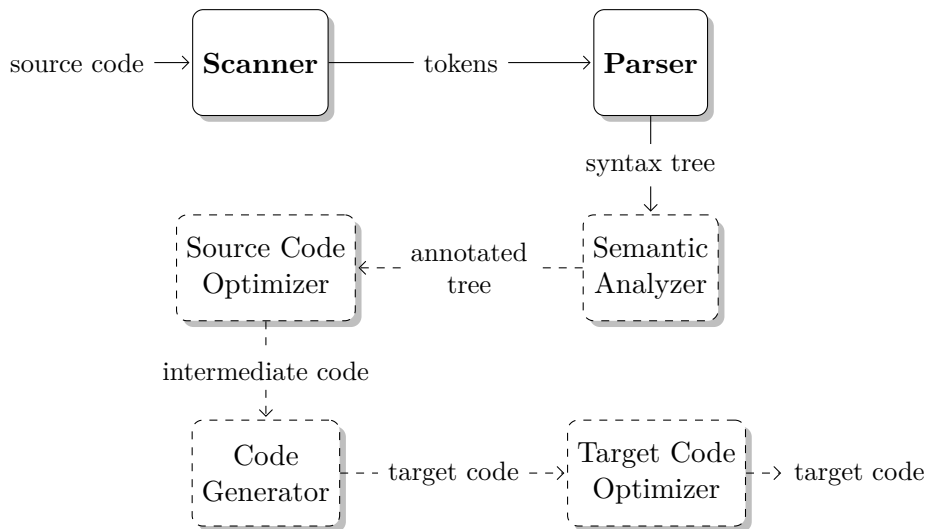
Figure 5.2: Interrupted compilation process. (Full compilation process borrowed from *Compiler construction: principles and practice* by Kenneth C. Louden [Lou97].)

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 automata. In fact, it is common to specify the tokens in regular expressions, that in turn is translated into a finite automata 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, that is usually specified in a context-free grammar, and often in a variant of the Backus–Naur Form[1]. 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. It only contain 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 that 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 fig. 5.3 on the following page.

Contrary to the Java Model, an abstract syntax tree is a heavy-weight representation of source code. It contains information about properties like

---

[1] https://en.wikipedia.org/wiki/Backus-Naur_Form

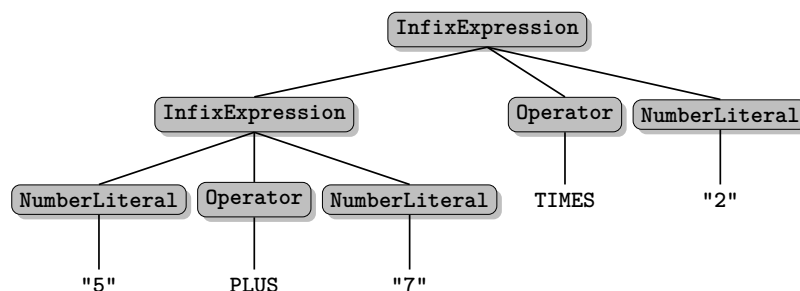type bindings for variables and variable bindings for names.

Figure 5.3: The abstract syntax tree for the expression `(5 + 7) * 2`.

5.2.1 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 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 a **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 types 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 fig. 5.4 on the next page.

Add more to the AST format tree? fig. 5.4 on the facing page
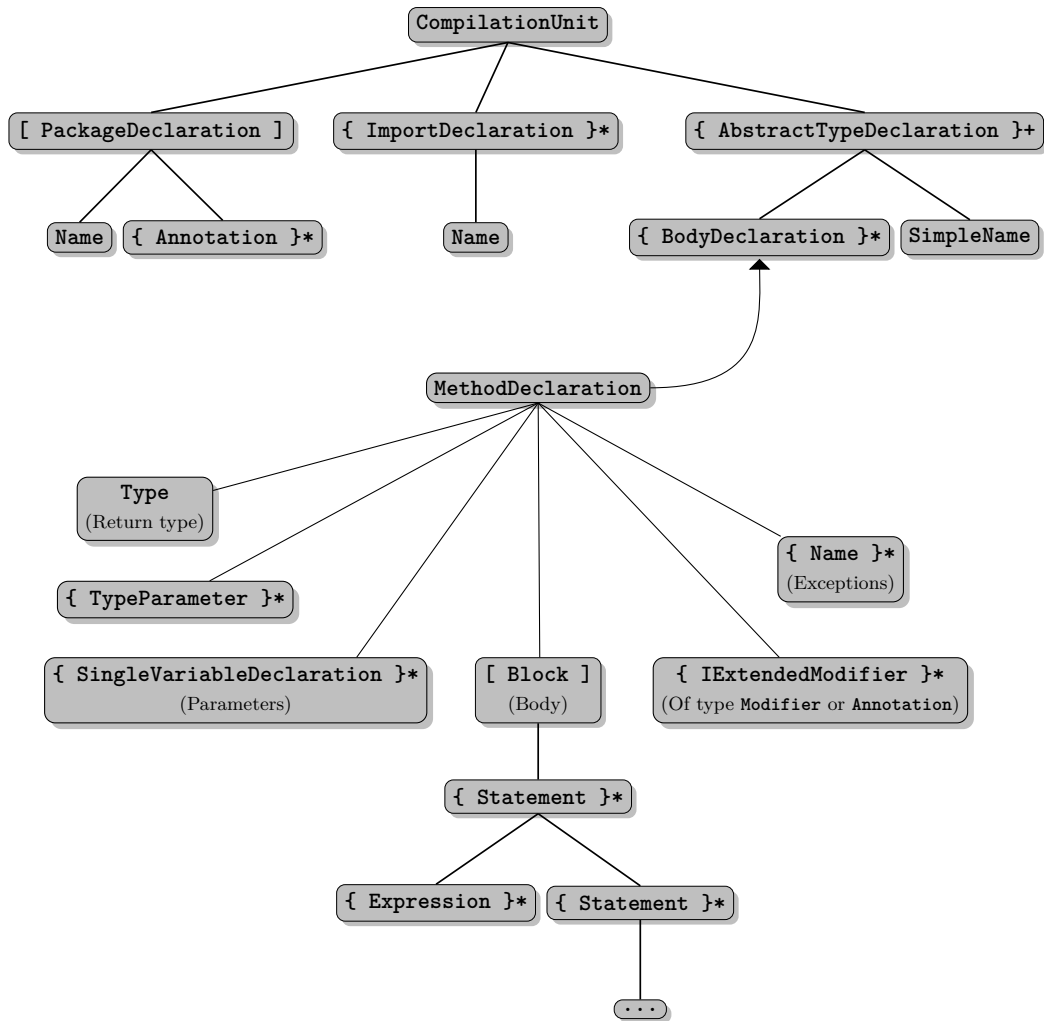
---

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

Figure 5.4: The format of the abstract syntax tree in Eclipse.

5.3 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, so we can conclude about the properties we are analysing. 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 provided 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

---

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

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 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 fig. 5.5 on the current page.
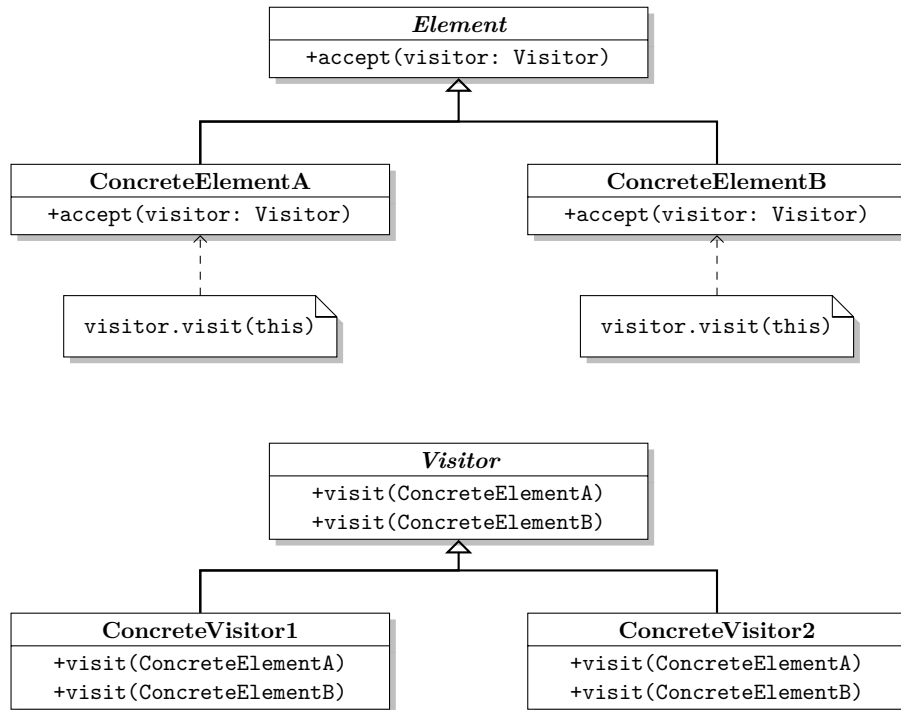

Figure 5.5: The Visitor Pattern.

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 of type `ASTNode` is very stable, but the functionality of its elements is extended every time someone needs to operate on the AST. Another aspect of the Eclipse implementation is that it is a public API, and 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 has **endVisit** methods for each node type, that 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** is also altering the behavior of **preVisit**, since the default implementation is responsible for calling it.

An example of a trivial **ASTVisitor** is shown in listing 6 on this page.

```java
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 6: 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**.

5.4 Property collectors

The prefixes and unfixes are found by property collectors[1]. A property collector is of the **ASTVisitor** type, and thus visits nodes of type **ASTNode** of the abstract syntax tree (see section 5.3 on page 37).

5.4.1 The PrefixesCollector

The **PrefixesCollector**[2] finds prefixes that makes up the basis for calculating move targets for the *Extract and Move Method* refactoring. It visits expression statements[3] and creates prefixes from its expressions in the case of method invocations. The prefixes found is registered with a prefix set, together with all its sub-prefixes.

---

[1]`no.uio.ifi.refaktor.extractors.collectors.PropertyCollector`
[2]`no.uio.ifi.refaktor.extractors.collectors.PrefixesCollector`
[3]`org.eclipse.jdt.core.dom.ExpressionStatement`

5.4.2 The UnfixesCollector

The `UnfixesCollector`[1] finds unfixes within a selection. That is prefixes that cannot be used as a basis for finding a move target in a refactoring.

An unfix can be a name that is assigned to within a selection. The reason that this cannot be allowed, is that the result would be an assignment to the `this` keyword, which is not valid in Java (see section 7.1 on page 51).

Prefixes that originates from variable declarations within the same selection are also considered unfixes. This is because when a method is moved, it needs to be called through a variable. If this variable is also within the method that is to be moved, this obviously cannot be done.

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

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

Interfaces types are not 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.) 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 implementation of the *Extract and Move Method* refactoring. 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 case of local classes, the problem is that, in the intermediate step, a selection referencing a local class would need to take the local class as a parameter if it were to be extracted to a new method. 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. 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.

The last class of names that are considered unfixes is names used in null tests. These are tests that reads 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 not be meaningful, since `this` would never be `null`.

---

[1] `no.uio.ifi.refaktor.extractors.collectors.UnfixesCollector`

```
// Before                          // After Extract Method
void declaresLocalClass() {        void declaresLocalClass() {
  class LocalClass {                 class LocalClass {
    void foo() {}                      void foo() {}
    void bar() {}                      void bar() {}
  }                                  }

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

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

Listing 7: When *Extract and Move Method* tries to use a variable with a local type as the move target, an intermediate step is taken that is not allowed. Here: **LocalClass** is not in the scope of **fooBar** in its intermediate location.

5.4.3 The ContainsReturnStatementCollector

The **ContainsReturnStatementCollector**[1] is a very simple property collector. It only visits the return statements within a selection, and can report whether it encountered a return statement or not.

5.4.4 The LastStatementCollector

The **LastStatementCollector**[2] collects the last statement of a selection. It does so by only visiting the top level statements of the selection, and compares the textual end offset of each encountered statement with the end offset of the previous statement found.

5.5 Checkers

The checkers are a range of classes that checks that text selections complies with certain criteria. All checkers operates under the assumption that the code they check is free from compilation errors. If a **Checker**[3] 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

---

[1] **no.uio.ifi.refaktor.analyze.collectors.ContainsReturnStatementCollector**
[2] **no.uio.ifi.refaktor.analyze.collectors.LastStatementCollector**
[3] **no.uio.ifi.refaktor.analyze.analyzers.Checker**

thrown. Many of the checkers either extends the **PropertyCollector** or utilizes one or more property collectors to verify some criteria. The checkers registered with the **LegalStatementsChecker** are described next. They are run in the order presented below.

5.5.1 The EnclosingInstanceReferenceChecker

The purpose of this checker is to verify that the names in a selection is not referencing any enclosing instances. This is for making sure that all references is legal in a method that is to be moved. Theoretically, some situations could be easily solved my passing a reference to the referenced class with the moved method (e.g. when calling public methods), but the dependency on the **MoveInstanceMethodProcessor** prevents this.

The **EnclosingInstanceReferenceChecker**[1] is a modified version of the **EnclosingInstanceReferenceFinder**[2] from the **MoveInstanceMethodProcessor**. Wherever the **EnclosingInstanceReferenceFinder** would create a fatal error status, the checker throws a **CheckerException**.

It works by first finding all of the enclosing types of a selection. Thereafter it visits all its simple names 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 is qualified with any name.

5.5.2 The ReturnStatementsChecker

The checker for return statements is meant to verify that a if a text selection contains a return statement, then every possible execution path within the selection ends in a return 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 the call to it would also be a valid return point for the calling method.

The property description implies, that if the selection is free from return statements, then the checker validates. So this is the first thing the checker investigates.

If the checker proceedes 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 is one of either a return or a throw statement. If the last statement of the selection is a return statement, then all execution paths within the selection should end in either this, or another, return statement. This is also true for a throw statement, since it causes an immediate exit from the current block, together with all outer blocks in its control flow that does not catch the thrown exception.

Return statements can be either explicit or implicit. An *explicit* return statement is formed by using the **return** keyword, while an *implicit* return statement is a statement that is not formed by the **return** keyword, but it

---

[1]`no.uio.ifi.refaktor.analyze.analyzers.EnclosingInstanceReferenceChecker`
[2]`org.eclipse.jdt.internal.corext.refactoring.structure.MoveInstanceMethodProcessor.EnclosingIn`

must be the last statement of a method that can have any side effects. This can happen in methods with a void return type.

To examine a selection for implicit return statements, the checker first finds the last statement declared in the enclosing method. If this statement is also enclosed by the selection, an implicit return statement is found. If the last statement of the method is not enclosed by the selection, then the checker does a search to see if the last statement of the selection 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 statement, that in turn is the last statement enclosed by 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 accepted parent statements. If the search ends in a method declaration, then the statement is considered to be the last reachable statement of the method. The block statements of loops are also accepted in this context, although their last containing statement cannot be considered an implicit return. This is handled later. If the last statement of a text selection is a either an explicit or implicit return statement, or it is a throw statement, the return statements checker validates.

So far, the properties checked results in the checker being interrupted if they are satisfying. So they validates the checker if something is *good*. The validation of the next two properties results in a `CheckerException` being thrown if they do not validate. So this form of checking invalidates the checker if something is *bad*.

> finish. . .

> Make flowchart of the previous section? (Change implementation/use control flow graph?)

5.5.3 The AmbiguousReturnValueChecker

This checker verifies that there are no *ambiguous return statements* in a selection. The problem with ambiguous return statements arise when a selection is chosen to be extracted into a new method, but it needs to return more than one value from that method. This problem occurs in two situations. The first situation arise 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 occur when there is only one such assignment, but there is also one or more return statements in the 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 collects whether there exists a return statement 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 5.5.2 on the preceding 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.

5.5.4   The IllegalStatementsChecker

This checker is designed to check for illegal statements.

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 is 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. 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 statement, and check that it is contained in 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 is allowed: Assignment to a non-final variable and assignment to an array access. All other assignments is regarded illegal.

Finish. . .

Chapter 6

Benchmarking

> Better name than "benchmarking"?

This part of the master 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 4.2.3 on page 27) over a larger software project, both to test its robustness but also its effect on different software metrics.

6.1 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 a with a fully functional Eclipse workbench. Most importantly, this gives us access to the Java Model of Eclipse (see section 5.1 on page 33).

6.1.1 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 require the absolute path to the project description file. It 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

---

[1] **no.uio.ifi.refaktor.benchmark.ProjectImporter**
[2] https://stackoverflow.com/questions/12401297

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.

6.2   Statistics

Statistics for the analysis and changes is captured by the `StatisticsAspect`[1]. This an *aspect* written in AspectJ.

6.2.1   AspectJ

AspectJ[2] 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 spans 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. 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 [14]. It is a function that modifies another function when the 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 8 on the facing page.

6.2.2   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.

---

[1] `no.uio.ifi.refaktor.aspects.StatisticsAspect`
[2] http://eclipse.org/aspectj/

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

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

Listing 8: An example of a pointcut named **methodAnalyze**, and an advice defined to be applied after it has occurred.

The **Statistics** class also include 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.

6.2.3 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 is found a selection that is a candidate for the refactoring, and for how many methods there is 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.

6.3 Optimizations

When looking for optimizations to make for the benchmarking process, I used the VisualVM[1] for the Java Virtual Machine to both profile the application and also to make memory dumps of its heap.

6.3.1 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 towards. 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

---

[1]http://visualvm.java.net/

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 was run.

The caching is managed by a cache manager, that 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 whenever there exists 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 as 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 9 on the current page. For a more thorough explanation of weak references in general, see [Nic06].

```
// 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 9: Showing the basic usage of soft references. Weak references is used the same way. (The references are part of the `java.lang.ref` package.)

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.

6.3.2   Memento

Write

Chapter 7

Eclipse Bugs Found

Add other things and change headline?

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

This bug[1] was found when analyzing what kinds of names that was to be considered as *unfixes* (see section 5.4.2 on page 40).

7.1.1 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.

7.1.2 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.

7.2 Eclipse bug 429416: IAE when moving method from anonymous class

I discovered[2] this bug during a batch change on the **org.eclipse.jdt.ui** project.

---

[1] https://bugs.eclipse.org/bugs/show_bug.cgi?id=420726
[2] https://bugs.eclipse.org/bugs/show_bug.cgi?id=429416

7.2.1   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 want to pass a `this` expression. The execution ends in an `IllegalArgumentException`[1] in `SimpleName`[2] and its `setIdentifier(String)` method. The simple name is attempted created in the method `createInlinedMethodInvocation`[3] 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. 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`[4] it all ends in an `IllegalArgumentException`.

7.2.2   How I solved the problem

Since the `MoveInstanceMethodProcessor` is instantiated in the `MoveMethod-RefactoringExecutor`[5], and only need to be a `MoveProcessor`[6], 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`[7]. 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.

7.3   Eclipse bug 429954: Extracting statement with reference to local type breaks code

The bug[8] was discovered when doing some changes to the way unfixes is computed.

---

[1]`java.lang.IllegalArgumentException`

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

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

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

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

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

[7]`no.uio.ifi.refaktor.refactorings.processors.ModifiedMoveInstanceMethodProcessor`

[8]https://bugs.eclipse.org/bugs/show_bug.cgi?id=429954

7.3.1 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 7 on page 41, but there in another setting.

7.3.2 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. So names representing variables of local types is considered unfixes (see section 5.4.2 on page 40).

> write more when fixing this in legal statements checker

Chapter 8

Related Work

8.1 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 1.9 on page 10). 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 help in figuring out what goes wrong if one should choose to op 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 one. They claim to have found evidence of that the *compositional paradigm* outperforms the *wizard-based.* It does so by reducing automation, which seem counterintuitive. Therefore they ask the question "What is an appropriate level of automation?", and thus questions what they feel is a rush toward more automation in the software engineering community.

---

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

Glossary

1. **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. 2

2. ***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. 7

3. ***Extract Method*** The *Extract Method* refactoring 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. 3

4. ***Move Method*** The *Move Method* refactoring is used to move a method from one class to another. This is useful if the method is using more features of another class than of the class which it is currently defined. Then all calls to this method must be updated, or the method must be copied, with the old method delegating to the new method. 6

5. **profiling** is to run a computer program through a profiler/with a profiler attached. 10

6. **software obfuscation** makes source code harder to read and analyze, while preserving its semantics. 2

7. **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]. 13

Bibliography

[11]   *JAVA EE Productivity Report 2011.* Survey. 2011. URL: http:
            //zeroturnaround.com/wp-content/uploads/2010/11/Java_EE_
            Productivity_Report_2011_finalv2.pdf.

[14]   *Advice (programming).* In: *Wikipedia, the free encyclopedia.*
            Page Version ID: 462233199. Mar. 14, 2014. URL: https://en.
            wikipedia.org/w/index.php?title=Advice_(programming)&oldid=
            462233199 (visited on 03/21/2014).

[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. ISSN: 0098-5589.
            DOI: 10.1109/32.295895.

[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).

[Fow99]   Martin Fowler. *Refactoring: improving the design of existing
            code.* Reading, MA: Addison-Wesley, 1999. ISBN: 0201485672.

[Gam+95]   Erich Gamma et al. *Design patterns: elements of reusable
            object-oriented software.* Reading, MA: Addison-Wesley, 1995.
            ISBN: 0201633612.

[Ker05]   Joshua Kerievsky. *Refactoring to patterns.* Boston: Addison-
            Wesley, 2005. ISBN: 0321213351.

[Lou97]   Kenneth C Louden. *Compiler construction: principles and
            practice.* Boston: PWS Pub. Co., 1997. ISBN: 0534939724
            9780534939724.

[MC09]   Robert C Martin and James O Coplien. *Clean code: a handbook of agile software craftsmanship*. Upper Saddle River, NJ [etc.]: Prentice Hall, 2009. ISBN: 9780132350884 0132350882.

[Mey88]   Bertrand Meyer. *Object-oriented software construction*. Prentice-Hall, 1988. ISBN: 0136290493 9780136290490 0136290310 9780136290315.

[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. ISSN: 1939-1471(Electronic);0033-295X(Print). DOI: 10.1037/h0043158.

[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).

[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. ISSN: 1074-3227.

[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.

Todo list