UiO : Department of Informatics
University of Oslo

Refactoring

An essay

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

1 What is Refactoring? 1
- 1.1 Defining refactoring . . . 1
- 1.2 The etymology of 'refactoring' . . . 2
- 1.3 Motivation – Why people refactor . . . 3
- 1.4 The magical number seven . . . 4
- 1.5 Notable contributions to the refactoring literature . . . 6
- 1.6 Tool support (for Java) . . . 6
- 1.7 The relation to design patterns . . . 8
- 1.8 The impact on software quality . . . 9
  - 1.8.1 What is software quality? . . . 9
  - 1.8.2 The impact on performance . . . 9
- 1.9 Composite refactorings . . . 10
- 1.10 Manual vs. automated refactorings . . . 10
- 1.11 Correctness of refactorings . . . 11
- 1.12 Refactoring and the importance of testing . . . 13
  - 1.12.1 Testing the code from correctness section . . . 13
- 1.13 The project . . . 14
- 1.14 Software metrics . . . 14

2 ... 15
- 2.1 The problem statement . . . 15
- 2.2 Choosing the target language . . . 15
- 2.3 Choosing the tools . . . 15
- 2.4 Organizing the project . . . 16
- 2.5 Continuous integration . . . 16
  - 2.5.1 Problems with AspectJ . . . 17

3 Refactorings in Eclipse JDT: Design, Shortcomings and Wishful Thinking 19
- 3.1 Design . . . 19
  - 3.1.1 The Language Toolkit . . . 19
- 3.2 Shortcomings . . . 20
  - 3.2.1 Absence of Generics in Eclipse Source Code . . . 21
  - 3.2.2 Composite Refactorings Will Not Appear as Atomic Actions . . . 21
- 3.3 Wishful Thinking . . . 21

4 Composite Refactorings in Eclipse **23**
4.1 A Simple Ad Hoc Model . . . 23
4.1.1 A typical `RefaktorChanger` . . . 23
4.2 The Extract and Move Method Refactoring . . . 23
4.2.1 The Building Blocks . . . 23
4.2.2 The ExtractAndMoveMethodChanger . . . 24
4.2.3 The SearchBasedExtractAndMoveMethodChanger . . . 26
4.2.4 Finding the IMethod . . . 27
4.2.5 The Prefix Class . . . 27
4.2.6 The PrefixSet Class . . . 27
4.2.7 Hacking the Refactoring Undo History . . . 27

5 Analyzing Source Code in Eclipse **29**
5.1 The Java model . . . 29
5.2 The Abstract Synax Tree . . . 30
5.2.1 The AST in Eclipse . . . 32
5.3 The ASTVisitor . . . 33
5.4 Property collectors . . . 35
5.4.1 The PrefixesCollector . . . 35
5.4.2 The UnfixesCollector . . . 36
5.4.3 The ContainsReturnStatementCollector . . . 37
5.4.4 The LastStatementCollector . . . 37
5.5 Checkers . . . 37
5.5.1 The EnclosingInstanceReferenceChecker . . . 38
5.5.2 The ReturnStatementsChecker . . . 38
5.5.3 The AmbiguousReturnValueChecker . . . 38
5.5.4 The IllegalStatementsChecker . . . 39

6 Benchmarking **41**
6.1 The benchmark setup . . . 41
6.1.1 The ProjectImporter . . . 41
6.2 Statistics . . . 42
6.2.1 AspectJ . . . 42
6.3 Optimizations . . . 42
6.3.1 Caching . . . 42
6.3.2 Memento . . . 42

7 Eclipse Bugs Found **45**
7.1 Eclipse bug 420726: Code is broken when moving a method that is assigning to the parameter that is also the move destination . . . 45
7.1.1 The bug . . . 45
7.1.2 The solution . . . 45
7.2 Eclipse bug 429416: IAE when moving method from anonymous class . . . 45
7.2.1 The bug . . . 46
7.2.2 How I solved the problem . . . 46

7.3 Eclipse bug 429954: Extracting statement with reference to local type breaks code 46
7.3.1 The bug 47
7.3.2 Actions taken 47

8 Related Work **49**
8.1 The compositional paradigm of refactoring 49

List of Figures

1.1 The Extract Superclass refactoring 11

5.1 The Java model of Eclipse. "`{ SomeElement }*`" means `SomeElement` zero or more times. For recursive structures, "`...`" is used. 30
5.2 Interrupted compilation process. (Full compilation process borrowed from *Compiler construction: principles and practice* by Kenneth C. Louden [Lou97].) 31
5.3 The abstract syntax tree for the expression `(5 + 7) * 2`. . . 32
5.4 The format of the abstract syntax tree in Eclipse. 33
5.5 The Visitor Pattern. 34

List of Tables

5.1 The elements of the Java Model. Taken from http://www.vogella.com/tutorials/EclipseJDT/article.html 29

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 such a thing as *software obfuscation* is refactoring. Software obfuscation is to make source code harder to read and analyze, while preserving its semantics. It could be done composing many, more or less randomly chosen, refactorings. Then the question arise whether it can be called a *composite refactoring* (see section 1.9 on page 10) or not? The answer is not obvious. First, there is no way to describe *the* mechanics of software obfuscation, beacause 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. The last point is important, since one of the motivations behind defining different refactorings is to build up a vocabulary for software professionals to reason and discuss about programs, similar to the motivation behind design patterns [Gam+95]. So for describing *software obfuscation*, it might be more appropriate to define what you do when performing it rather than precisely defining its mechanics in terms of other refactorings.

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[3] community, and the word "refactoring" is mentioned in a book by Leo Brodie, called *Thinking Forth* [Bro], first published in 1984[4]. The exact word is only printed one place [Bro, 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. [Bro, p. 172]

Brodie also express what *factoring* means to him:

[1] *Smalltalk*, object-oriented, dynamically typed, reflective programming language. See http://www.smalltalk.org

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

[3] *Forth* – stack-based, extensible programming language, without type-checking. See http://www.forth.org

[4] *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.

> Factoring means organizing code into useful fragments. To make a fragment useful, you often must separate reusable parts from non-reusable parts. The reusable parts become new definitions. The non-reusable parts become arguments or parameters to the definitions. [Bro, 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 [Gam+95] 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 already mentioned, people often refactor to get rid of duplication. Moving identical or similar code into methods, and maybe pushing methods up or down in their class hierarchies. Making template methods for overlapping algorithms/functionality and so on. It is all about gathering what belongs together and putting it all in one place. The resulting code is then easier to maintain. When removing the implicit coupling[1] between code snippets, the location of a bug is limited to only one place, and new functionality need only to be added to this one place, instead of a number of places people might not even remember.

A problem you often encounter when programming, is that a program contains a lot of long and hard-to-grasp methods. It can then help to break the methods into smaller ones, using the *Extract Method* refactoring [Fow99]. Then you may discover something about a program that you were not aware of before; revealing bugs you did not know about or could not find due to the complex structure of your program. Making the methods smaller and giving good names to the new ones clarifies the algorithms and enhances the *understandability* of the program (see section 1.4 on page 4). This makes refactoring an excellent method for exploring unknown program code, or code that you had forgotten that you wrote.

Proof?

Most primitive refactorings are simple. Their true power is first 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 be evolved 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.

[1] When duplicating code, the code might not be coupled in other ways than that it is supposed to represent 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 the multiple pieces of code.

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

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.

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. 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].

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

An example from the article addresses the problem of memorizing a sequence of binary digits. Let us say we have the following sequence[1] 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 latest sequence is the first sequence recoded to be represented by digits with 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 [Gam+95] 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 has code that is easier (and better) understood.

[1] The example presented here is slightly modified (and shortened) from what is presented in the original article [Mil56], but it is essentially the same.

1.5 Notable contributions to the refactoring literature

Update with more contributions

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 readers 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 page 8.)

1.6 Tool support (for Java)

This section will briefly compare the refatoring support of the three IDEs *Eclipse*[1], *IntelliJ IDEA*[2] and *NetBeans*[3]. These are the most popular Java IDEs [Jav].

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

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

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

[3] https://netbeans.org/

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, its default destination for the move is itself, although it refuses to perform the refactoring if chosen. But the worst part is, that if moving the method `f` of the class `C` to the class `X`, it will break the code. The result is shown in listing 1 on page 7.

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

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

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

NetBeans will try to make 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 seems to have fairly good support for primitive refactorings, but what about more complex ones, such as the *Extract Class* [Fow99]? The *Extract Class* refactoring works by creating a class, for then to move members to that class and access them from the old class via a reference to the new class. *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 (or withdrawn) 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 seem to have made an attempt on providing this refactoring. (Well, it probably has, but it does not show in the IDE.)

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 maintanable and extendable source code.

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

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

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

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

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

This can easily happen when relying largely on up-front design. Then it is natural, in the very beginning, to try to build in all the flexibility that one believes will be necessary throughout the lifetime of a software system. According to Joshua Kerievsky "That sounds reasonable — if you happen to be psychic." [Ker05, p. 1] He is advocating what he believes is a better approach: To let software continually evolve. To start with a simple design that meets today's needs, and tackle future needs by refactoring to satisfy them. He believes that this is a more economic approach than investing time and money into a design that inevitably is going to change. By relying on continuously refactoring a system, its design can be made simpler without sacrificing flexibility. To be able to fully rely on this approach, it is of utter importance to have a reliable suit of tests to lean on. (See section 1.12 on page 13.) 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

[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.

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

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

So, although an increased amount of method calls could potentially slow down programs, one should avoid premature optimization and sacrificing good design, leaving the performance tuning until after profiling[1] 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. For an overview of the *Extract Superclass* refactoring, see fig. 1.1 on page 11.

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

[1] For and example of a Java profiler, check out VisualVM: http://visualvm.java.net/

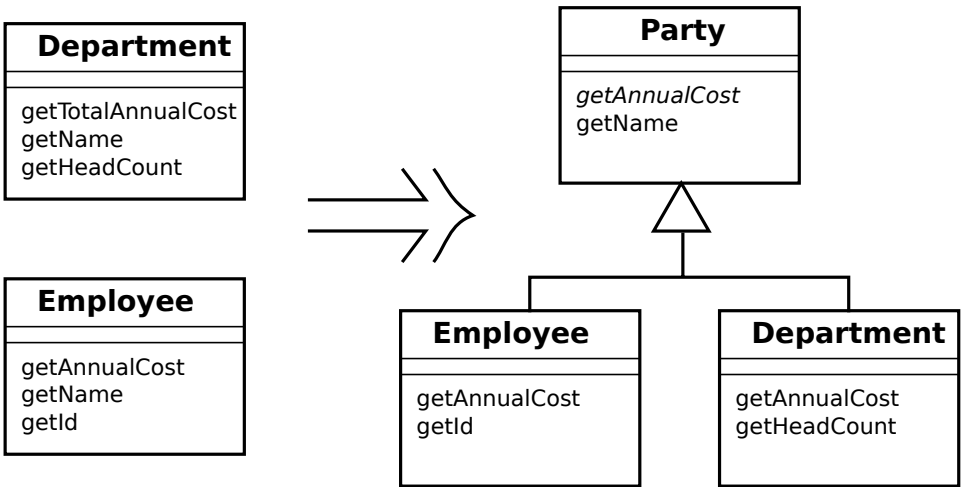

Figure 1.1: The Extract Superclass refactoring

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.

Take for instance the renaming of classes, methods and fields. For complex programs these refactorings are almost impossible to get right. Attacking them with textual search and replace, or even regular expressions, will fall short on these tasks. Then it is crucial to have proper tool support that can perform them automatically. Tools that can parse source code and thus have semantic knowledge about which occurrences of which names belong to what construct in the program. For even trying to perform one of these complex task manually, one would have to be very confident on the existing test suite (see section 1.12 on page 13).

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. I will now present an example of an *Extract Method* refactoring followed by a *Move Method* refactoring that breaks a program in both the *Eclipse* and *IntelliJ* IDEs[1]. The following piece of code shows the target for the composed refactoring:

[1]The NetBeans IDE handles this particular situation without altering the program's beavior, mainly because its Move Method refactoring implementation is a bit flawed in other ways (see section 1.6 on page 6).

```
1  public class C {
2      public X x = new X();
3
4      public void f() {
5          x.m(this);
6          x.n();
7      }
8  }
```

The next piece of code shows the destination of the refactoring. Note that the method `m(C c)` of class `C` assigns to the field `x` of the argument `c` that has type `C`:

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

The refactoring sequence works by extracting line 5 and 6 from the original class `C` into a method `f` with the statements from those lines as its method body. The method is then moved to the class `X`. The result is shown in the following two pieces of code:

```
1  public class C {
2      public X x = new X();
3
4      public void f() {
5          x.f(this);
6      }
7  }
```

```
1  public class X {
2      public void m(C c) {
3          c.x = new X();
4      }
5      public void n() {}
6      public void f(C c) {
7          m(c);
8          n();
9      }
10 }
```

After the refactoring, the method `f` of class `C` is calling the method `f` of class `X`, and the program now behaves different than before. (See line 5 of the version of class `C` after the refactoring.) Before the refactoring, the methods `m` and `n` of class `X` are called on different object instances (see line 5 and 6 of the original class `C`). After, they are called on the same object,

and the statement on line 3 of class `X` (the version after the refactoring) no longer have any effect in our example.

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]

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 confindent 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.

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

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?

1.13 The project

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.

1.14 Software metrics

Is this the appropriate place to have this section?

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

Chapter 2

• • •

write

2.1 The problem statement

2.2 Choosing the target language

Choosing which programming language to use as the target for manipulation is not a very difficult task. The language has to be an object-oriented programming language, and it must have existing tool support for refactoring. The *Java* programming language[1] is the dominating language when it comes to examples 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[2] dozens of other programming languages, 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 language of the author of this thesis.

2.3 Choosing the tools

When choosing a tool for manipulating Java, there are certain criterias 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,

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

[2] They compile to java bytecode.

that are commited to answering questions regarding the use and misuse of the products, that to a large degree is made by the cummunity itself.

There is a certain class of tools that meet these criterias, namely the class of *IDEs*[1]. These are proagrams that is ment to support the whole production cycle of a cumputer 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 popular [Jav] 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*[2].

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*[3] has been set up for the project[4]. 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*[5] 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.

[1] *Integrated Development Environment*
[2] http://git-scm.com/
[3] http://jenkins-ci.org/
[4] A work mostly done by the supervisor.
[5] http://www.eclipse.org/buckminster/

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

I then managed to write an *Ant*[3] 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://www.eclipse.org/ajdt/

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

[3] 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, 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`[1] and `Change`[2]. (There is 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.)

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

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

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.

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.

3.2 Shortcomings

refine

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.

[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`

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 has 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 occur in the middle of a sequence of refactorings, it can leave the project in a state where the composite refactoring was executed only partly. 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 4.2.7 on page 27.)

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 **`ExtractAndMoveMethodAnalyzer`**[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 37). 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 candidates 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, se section 5.4 on page 35.) 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 **`MoveMethodRefactoringExecutor`** with the resources needed after executing the extract method refactoring. (See section 4.2.4 on page 27.)

[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.

Elaborate?

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

Write...

4.2.4 Finding the IMethod

Rename section. Write??

4.2.5 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.

4.2.6 The PrefixSet Class

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

There are two ways 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.

4.2.7 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[3] in the history of the refactorings.

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

[1] `java.util.HashMap`
[2] `java.util.Set`
[3] `org.eclipse.ltk.core.refactoring.Change`
[4] `org.eclipse.ltk.core.refactoring.IUndoManager`
[5] `org.eclipse.ltk.core.refactoring.CompositeChange`

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[1] 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[2]. 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[3]. 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.

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.

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

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 page 29.

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 page 30.

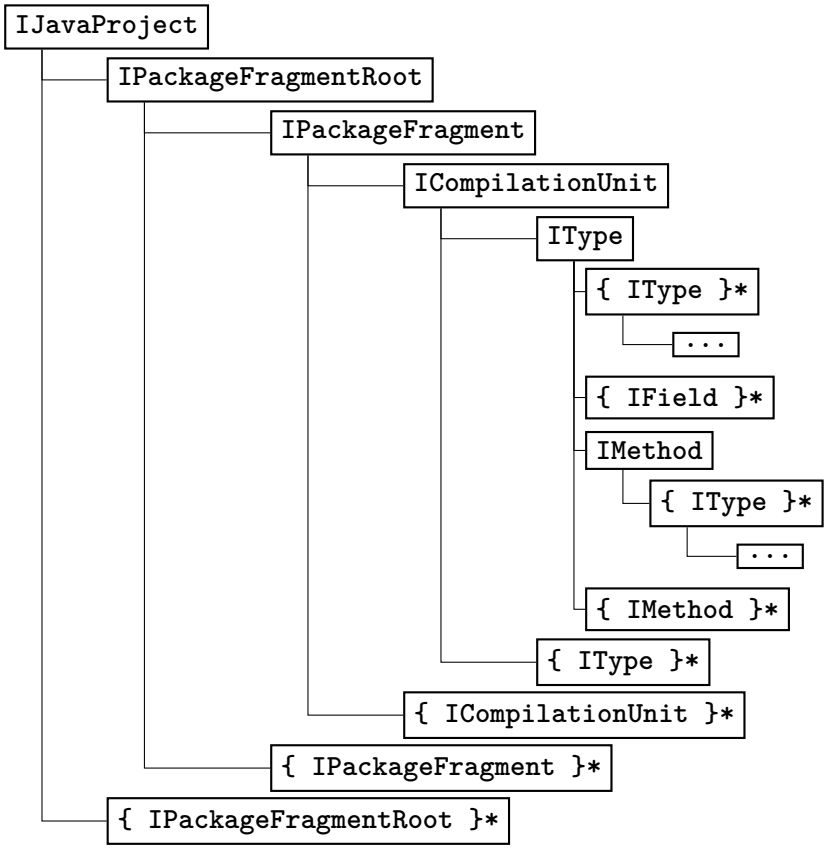

Figure 5.1: The Java model of Eclipse. "`{ SomeElement }*`" means `SomeElement` zero or more times. For recursive structures, "`...`" is used.

5.2 The Abstract Synax Tree

Eclipse is following the common paradigm of using an abstract syntaxt 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 anayzes 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 page 31).

The process starts with a *scanner*, or lexer. The job of the scanner is to read the source code and divide it into tokens for the parser. Therefore, it is also sometimes called a tokenizer. A token is a logical unit, defined in the language specification, consisting of one or more consecutive characters. In the java language the tokens can for instance be the `this` keyword, a curly bracket `{` or a `nameToken`. It is recognized by the scanner on the basis

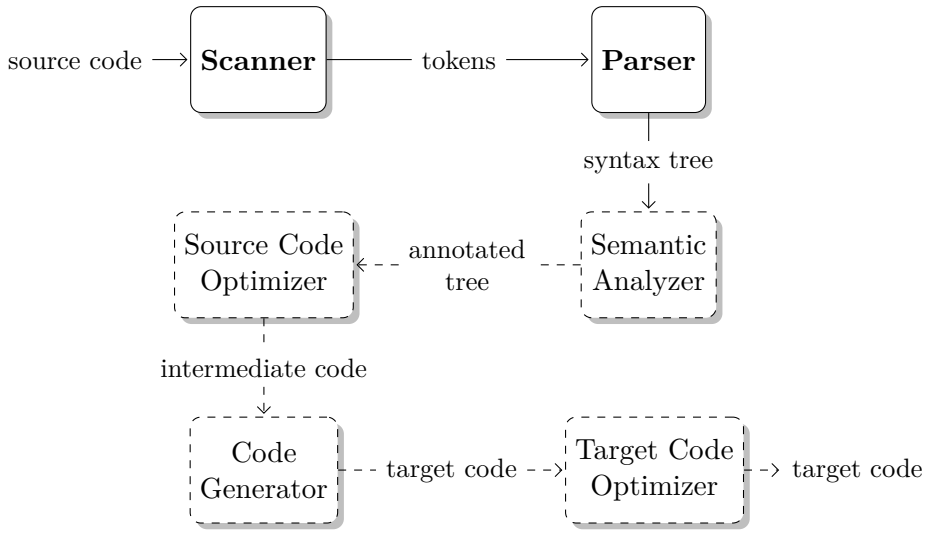

Figure 5.2: Interrupted compilation process. (Full compilation process borrowed from *Compiler construction: principles and practice* by Kenneth C. Louden [Lou97].)

of something eqivalent 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 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 page 32.

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

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

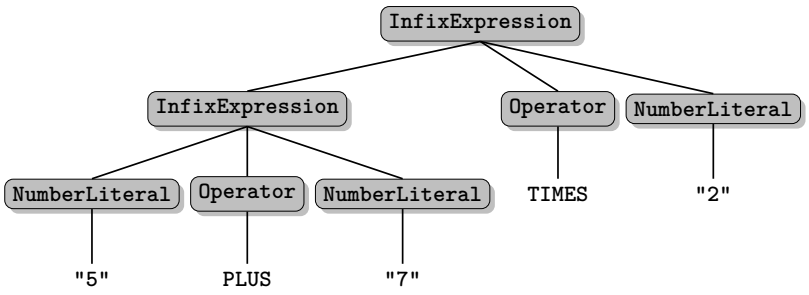

Figure 5.3: The abstract syntax tree for the expression `(5 + 7) * 2`.

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 **`AnnotationTypeDeclaration`**, **`EnumDeclaration`** or **`TypeDeclaration`**. The children of an **`AbstractTypeDeclaration`** must be a subtype of a **`BodyDeclaration`**. These subtypes are: **`AnnotationTypeMemberDeclaration`**, **`EnumConstantDeclaration`**, **`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 page 33.

Add more to the AST format tree? fig. 5.4 on page 33

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

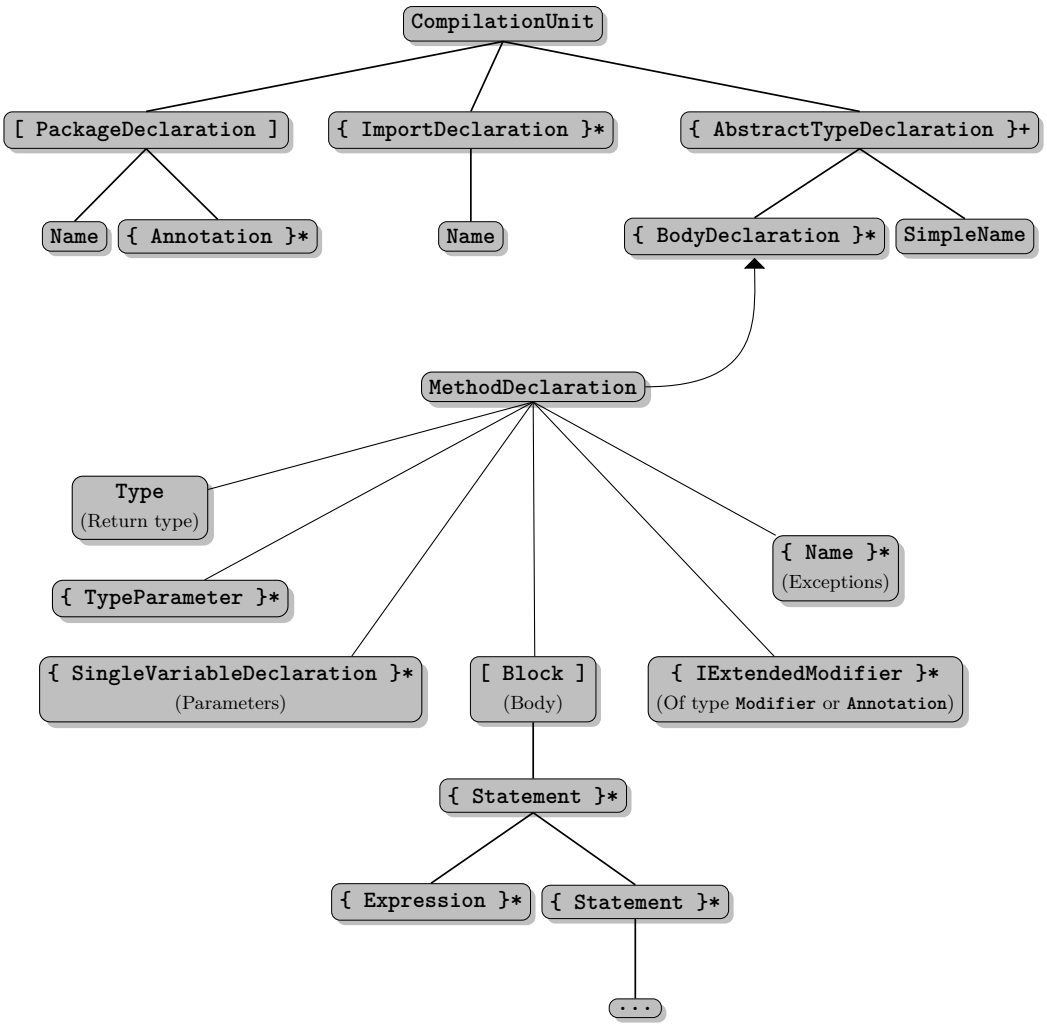

Figure 5.4: The format of the abstract syntax tree in Eclipse.

5.3 The ASTVisitor

So far, the only thing that has been adressed is how the 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 page 34.

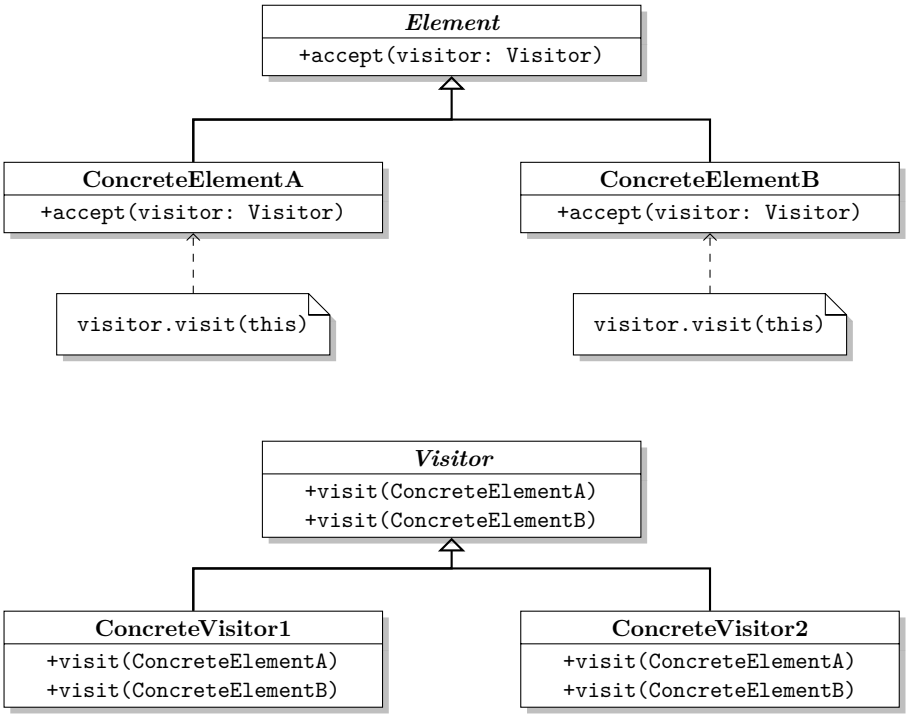

Figure 5.5: The Visitor Pattern.

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 cas 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 2 on page 35.

```
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 2: 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 33).

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 45).

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

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

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

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

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

Listing 3: 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 encuntered statement with the end offset of the previous statement found.

5.5 Checkers

The checkers are a range of classes that checks that selections complies with certian criterias. 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 thrown. Many of the checkers either extends

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

the `PropertyCollector` or utilizes one or more property collectors to verify some criterias. 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

Write. . . /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 page 38).

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 the collected keys. If more than one unique referral is found, or only one referral is found, but the selection also

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

[2] `org.eclipse.jdt.internal.corext.refactoring.structure.MoveInstanceMethodProcessor.EnclosingIn`

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 26) over a larger software project, both to test its roubustness 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 29).

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 unconvenient 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, and thus does not belong naturally in any of them. It can also be functionality that does not concern the business logic of an application, and thus may be a burden when entangled in parts of the source code it does not really belong. Examples include logging, debugging, optimizations and security.

Aspects is interacting with other modules by defining advices. 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 or when there is thrown an exception in the pointcut. 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 4 on page 43.

6.3 Optimizations

6.3.1 Caching

6.3.2 Memento

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

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 36).

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 clas 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 `MoveMethodRefactoringExecutor`[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 3 on page 37, 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 36).

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 ment 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

Bibliography

[Bro] Leo Brodie. *Thinking Forth*. 1984, 1994, 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).

[Fow01] Martin Fowler. *Crossing Refactoring's Rubicon*. 2001. URL: http://martinfowler.com/articles/refactoringRubicon.html.

[Fow03] Martin Fowler. *Etymology Of Refactoring*. 2003. URL: http://martinfowler.com/bliki/EtymologyOfRefactoring.html.

[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.

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

[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.

[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

2do i
2do i
Proof? 3
2do 6
2do 7
motivation, examples, manual vs automated?, what about refactoring in a very large code base? 10
2do 14
2do 14
2do 15
What about the language specific part? 19
refine 20
2do 21
2do 25
Elaborate? 26
2do 26
2do 27
2do 27
2do 32
2do 38
2do 39
2do 41
2do 45
2do 47