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

List of Figures

List of Tables

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 [6], 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. [6, 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 [7]. 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 [5], 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* [14], published in 1997. Allegedly [5], 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* [1], first published in 1984[4]. The exact word is only printed one place [1, 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. [1, 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. [1, 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 [7] 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 [6]. 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* [7] 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* [12] 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 [6].

> . . . recoding is an extremely powerful weapon for increasing the amount of information that we can deal with. [12, 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 [7] 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. [12, 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 [12], 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* [13]. 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* [6]. This is maybe the most influential text on refactoring. It bares similarities with Opdykes thesis [13] 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* [9]. This book is heavily influenced by Fowler's *Refactoring* [6] and the "Gang of Four" *Design Patterns* [7]. 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 [8].

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 [4, 15].

Regarding the *Move Method* refactoring, the *Eclipse* and *IntelliJ* IDEs do the job in very similar manners. In most situations they both do a satisfying job by producing the expected outcome. But they do nothing

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

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

[3]https://netbeans.org/

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* [6]? 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* [10] 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. [7, p. 353]

Although sometimes associated with over-engineering [6, 9], 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." [11] 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. [6, 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. [9, 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.” [9, 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. [6, 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 [3] disproves this view in the case of polymorphism. He did an experiment on, what he calls, “Transform Self Type Checks” where you introduce a new polymorphic method and a new class hierarchy

[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.

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. [6, 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 [6], 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 [6]. In its simplest form, it is composed of the previously described primitive refactorings, in addition to the *Pull Up Constructor Body* refactoring [6]. 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 program's behavior is a refactoring. For small refactorings, such as *Extract*

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

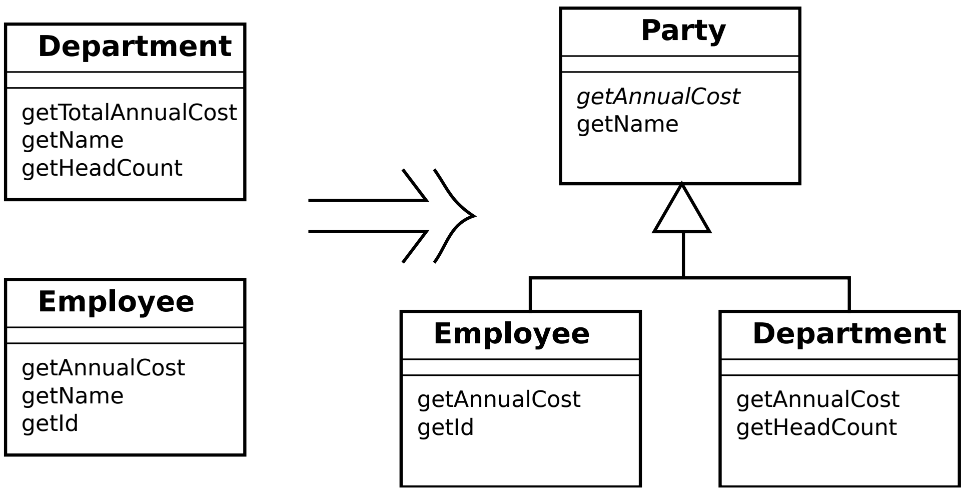

Figure 1.1: The Extract Superclass refactoring

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 ther program's beavior, mainly because its Move Method refactoring implementation is a bit rancid 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. [6]

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.

Write …

Assuming a sequential (non-concurrent) program:

```
tracematch (C c, X x) {
  sym m before:
    call(* X.m(C)) && args(c) && cflow(within(C));
  sym n before:
    call(* X.n()) && target(x) && cflow(within(C));
  sym setCx after:
    set(C.x) && target(c) && !cflow(m);

  m n

  { assert x == c.x; }
}
```

1.13 The project

The aim of this 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 *Extract Method* and *Move Method* 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* [2].

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

[1] *Integrated Development Environment*

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 26.)

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 17, 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

[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`

Extract Method refactoring [6] and the *Move Method* refactoring [6]. 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 Class

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. 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. 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.

Clean up sections/subsections.

The `ExtractAndMoveMethodExecutor`

If the analysis finds a possible target for the composite refactoring, it is executed by an **`ExtractAndMoveMethodExecutor`**[4]. It is composed of the two executors known as **`ExtractMethodRefactoringExecutor`**[5] and **`MoveMethodRefactoringExecutor`**[6]. 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 4.2.3 on page 24.)

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.

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

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 Finding the IMethod

Rename section. Write.

4.2.4 The ExtractAndMoveMethodPrefixesExtractor Class

This extractor extracts properties needed for building the Extract and Move Method refactoring. It searches through the given selection to find safe prefixes, and those prefixes form a base that can be used to compute possible targets for the move part of the refactoring. It finds both the candidates, in the form of prefixes, and the non-candidates, called unfixes. All prefixes (and unfixes) are represented by a **Prefix**[1], and they are collected into prefix sets.[2].

The prefixes and unfixes are found by property collectors[3]. A property collector follows the visitor pattern [7] and is of the **ASTVisitor**[4] type. An **ASTVisitor** visits nodes in an abstract syntax tree that forms the Java document object model. The tree consists of nodes of type **ASTNode**[5].

The PrefixesCollector

The **PrefixesCollector**[6] is of type **PropertyCollector**. It visits expression statements[7] 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.

Rewrite in the case of changes to the way prefixes are found

The UnfixesCollector

The **UnfixesCollector**[8] 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 6.1 on page 31).

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

[1] `no.uio.ifi.refaktor.extractors.Prefix`
[2] `no.uio.ifi.refaktor.extractors.PrefixSet`
[3] `no.uio.ifi.refaktor.extractors.collectors.PropertyCollector`
[4] `org.eclipse.jdt.core.dom.ASTVisitor`
[5] `org.eclipse.jdt.core.do.ASTNode`
[6] `no.uio.ifi.refaktor.extractors.collectors.PrefixesCollector`
[7] `org.eclipse.jdt.core.dom.ExpressionStatement`
[8] `no.uio.ifi.refaktor.extractors.collectors.UnfixesCollector`

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.

Example?

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.

4.2.5 The Prefix Class

?

4.2.6 The PrefixSet Class

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[1] in the history of the refactorings.

My first impulse was to remove the, in this case, last two undo changes from the undo manager[2] for the Eclipse refactorings, and then add them to a composite change[3] that could be added back to the manager. The interface of the undo manager does not offer a way to remove/pop the last added undo change, so a possible solution could be to decorate [7] the undo manager, to intercept and collect the undo changes before delegating to the **`addUndo`** method[4] of the manager. Instead of giving it the intended undo change, a null change could be given to prevent it from making any changes if run. Then one could let the collected undo changes form a composite change to be added to the manager.

There is a technical challenge with this approach, and it relates to the

[1]`org.eclipse.ltk.core.refactoring.Change`
[2]`org.eclipse.ltk.core.refactoring.IUndoManager`
[3]`org.eclipse.ltk.core.refactoring.CompositeChange`
[4]`org.eclipse.ltk.core.refactoring.IUndoManager#addUndo()`

undo manager, and the concrete implementation UndoManager2[1]. 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[2]. 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.internal.core.refactoring.UndoManager2`

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

Chapter 5

Analyzing Code

5.1 AST

Explain what it is, or just how it is structured in Eclipse and how to analyze it?

5.2 Illegal selections

5.2.1 Not all branches end in return

5.2.2 Ambiguous return statement

This problem occurs when there is either more than one assignment to a local variable that is used outside of the selection, or there is only one, but there are also return statements in the selection.

Explain why we do not need to consider variables assigned inside local/anonymous classes. (The referenced variables need to be final and so on. . .)

Chapter 6

Eclipse Bugs

Add other things and change headline?

6.1 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*.

refer to unfixes

6.1.1 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.

6.1.2 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.

6.2 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

6.2.1 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 analyses 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`**.

6.2.2 How I solved the problem

Since the **`MoveInstanceMethodProcessor`** is instantiated in the **`MoveMethod-RefactoringExecutor`**[5], and only need to be a **`MoveProcessor`**[6], I was able to copy the code for the original move processor and modify it so that it works better for me. It is now called **`ModifiedMoveInstanceMethodProcessor`**[7]. The only modification done (in addition to some imports and suppression of warnings), is in the **`createInlinedMethodInvocation`**. When the declaring class of the method to move is anonymous, the **`this`** expression in the parameter list is not qualified with the declaring class' (empty) name.

[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`

Chapter 7

Related Work

7.1 The compositional paradigm of refactoring

This paradigm builds upon the observation of Vakilian et al. [16], 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

[1] Leo Brodie. *Thinking Forth.* 1984, 1994, 2004. URL: http://thinking-forth.sourceforge.net/.

[2] 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.

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

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

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

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

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

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

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

[10] 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.

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

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

Todo list