UiO : Department of Informatics
University of Oslo

Refactoring

An essay

Erlend Kristiansen
2014

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 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.

In the extreme case one could argue that software obfuscation is refactoring. It is often used to protect proprietary software. It restrains uninvited viewers, so they have a hard time analyzing code that they are

[1]The structure observable by the programmer.

not supposed to know how works. This could be a problem when using a language that is possible to decompile, such as Java.

Obfuscation could be done composing many, more or less randomly chosen, refactorings. Then the question arises whether it can be called a *composite refactoring* or not (see section 9 on page 10)? The answer is not obvious. First, there is no way to describe the mechanics of software obfuscation, because there are infinitely many ways to do that. Second, obfuscation can be thought of as *one operation*: Either the code is obfuscated, or it is not. Third, it makes no sense to call software obfuscation *a refactoring*, since it holds different meaning to different people.

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

2 The etymology of 'refactoring'

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

According to Martin Fowler [Fow03], there may also be more than one origin of the word. The most well-known source, when it comes to the origin of *refactoring*, is the Smalltalk[1] community and their infamous Refactoring Browser[2] described in the article *A Refactoring Tool for Smalltalk* [RBJ97], published in 1997. Allegedly [Fow03], the metaphor of factoring programs was also present in the Forth[1] community, and the word "refactoring" is mentioned in a book by Leo Brodie, called *Thinking Forth* [Bro04], first published in 1984[3]. The exact word is only printed one place [Bro04, p. 232], but the term *factoring* is prominent in the book, that also contains a whole chapter dedicated to (re)factoring, and how to keep the (Forth) code clean and maintainable.

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

Brodie also express what *factoring* means to him:

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

[1] Programming language

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

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

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.

3 Motivation – Why people refactor

There are many reasons why people want to refactor their programs. They can for instance do it to remove duplication, break up long methods or to introduce design patterns into their software systems. The shared trait for all these are that peoples' intentions are to make their programs *better*, in some sense. But what aspects of their programs are becoming improved?

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

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

Most primitive refactorings are simple, and usually involves moving code around [Ker05]. The motivation behind them may first be revealed when they are combined into larger — higher level — refactorings, called *composite refactorings* (see section 9 on page 10). Often the goal of such a series of refactorings is a design pattern. Thus the design can *evolve* throughout the lifetime of a program, as opposed to designing up-front. It is all about being structured and taking small steps to improve a program's design.

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

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

increasing the productivity in the software development process.

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

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

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

4 The magical number seven

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

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

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

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

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

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

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

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

5 Notable contributions to the refactoring literature

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

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

2005 Joshua Kerievsky: *Refactoring to Patterns* [Ker05]. This book is heavily influenced by Fowler's *Refactoring* [Fow99] and the "Gang of Four" *Design Patterns* [Gam+95]. It is building on the refactoring catalogue from Fowler's book, but is trying to bridge the gap between *refactoring* and *design patterns* by providing a series of higher-level composite refactorings, that makes code evolve toward or away from certain design patterns. The book is trying to build up the reader's intuition around *why* one would want to use a particular design pattern, and not just *how.* The book is encouraging evolutionary design (see section 7 on the next page).

6 Tool support (for Java)

This section will briefly compare the refactoring support of the three IDEs Eclipse[1], IntelliJ IDEA[2] and NetBeans[3]. These are the most popular Java IDEs [11].

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

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

Regarding the *Move Method* refactoring, the Eclipse and IntelliJ IDEs do the job in very similar manners. In most situations they both do a satisfying job by producing the expected outcome. But they do nothing to check that the result does not break the semantics of the program (see section 11 on page 11). The NetBeans IDE implements this refactoring in a somewhat unsophisticated way. For starters, the refactoring's default destination for the move, is the same class as the method already resides in, although it refuses to perform the refactoring if chosen. But the worst part is, that if moving the method `f` of the class `C` to the class `X`, it will break the code. The result is shown in listing 1 on the next page.

NetBeans will try to create code that call the methods `m` and `n` of `X` by accessing them through `c.x`, where `c` is a parameter of type `C` that is added

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

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

[3]https://netbeans.org/

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

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

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 11 on page 11.) If `c.x` for some reason is inaccessible to `X`, as in this case, the refactoring breaks the code, and it will not compile. NetBeans presents a preview of the refactoring outcome, but the preview does not catch it if the IDE is about break the program.

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

7 The relation to design patterns

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

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

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

> Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.[1]

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

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

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

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

This can easily happen when relying largely on up-front design. Then it is natural, in the very beginning, to try to build in all the flexibility that one believes will be necessary throughout the lifetime of a software system. According to Joshua Kerievsky “That sounds reasonable — if you happen 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 12 on

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

page 12). This makes the design process more natural and less characterized by difficult decisions that has to be made before proceeding in the process, and that is going to define a project for all of its unforeseeable future.

8 The impact on software quality

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.)

8.2 The impact on performance

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

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

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

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

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

[1]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.

9 Composite refactorings

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

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

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

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

An example of a composite refactoring is the *Extract Superclass* refactoring [Fow99]. In its simplest form, it is composed of the previously described primitive refactorings, in addition to the *Pull Up Constructor Body* refactoring [Fow99]. It works by creating an abstract superclass that the target class(es) inherits from, then by applying *Pull Up Field*, *Pull Up Method* and *Pull Up Constructor Body* on the members that are to be members of the new superclass. If there are multiple classes in play, their interfaces may need to be united with the help of some rename refactorings, before extracting the superclass. For an overview of the *Extract Superclass* refactoring, see fig. 1 on this page.

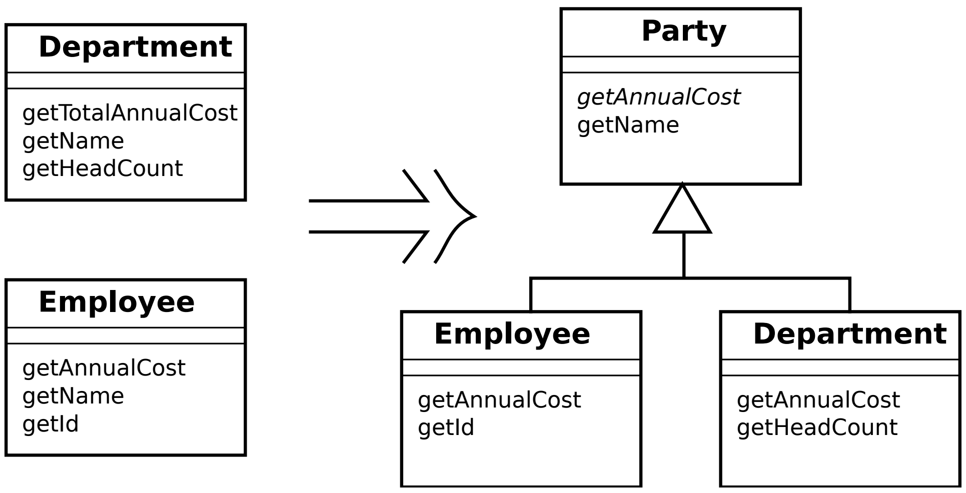

Figure 1: The Extract Superclass refactoring, with united interfaces.

10 Manual vs. automated refactorings

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

Consider the renaming of classes, methods and fields. For complex programs these refactorings are almost impossible to get right. Attacking them with textual search and replace, or even regular expressions, will fall short on these tasks. Then it is crucial to have proper tool support that can perform them automatically. Tools that can parse source code and thus have semantic knowledge about which occurrences of which names belong 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 12 on the next page).

11 Correctness of refactorings

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

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

```
// Refactoring target             // Method destination
public class C {                  public class X {
  public X x = new X();             public void m(C c) {
                                      c.x = new X();
  public void f() {                   // If m is called from
    x.m(this);                        // c, then c.x no longer
    // Not the same x                 // equals 'this'
    x.n();                          }
  }                                 public void n() {}
}                                 }
```

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

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

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

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

```
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      // Extracted and
7      // moved method
8      public void f(C c) {
9          m(c);
10         n();
11     }
12 }
```

Listing 3: The result of the composed refactoring.

The bug introduced in the previous example is of such a nature[1] that it is very difficult to spot if the refactored code is not covered by tests. It does not generate compilation errors, and will thus only result in a runtime error or corrupted data, which might be hard to detect.

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

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

will show, eventually. Then there are the *behavior-changing* errors. These errors are of the worst kind. They do not show up during compilation and they do not turn on a blinking red light during runtime either. The program can seem to work perfectly fine with them in play, but the business logic can be damaged in ways that will only show up over time.

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

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

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

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 11 on page 11. This example may be trivial, but the essence is clear. The only problem with the example is that it is not clear how to create automated tests for it, without changing it in intrusive ways.

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

This problem is still open.

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.

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

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.

Glossary

1. **design pattern** A design pattern is a named abstraction, that is meant to solve a general design problem. It describes the key aspects of a common problem and identifies its participators and how they collaborate. 2

2. ***Extract Class*** The *Extract Class* refactoring works by creating a class, for then to move members from another class to that class and access them from the old class via a reference to the new class. 7

3. ***Extract Method*** The *Extract Method* refactoring is used to extract a fragment of code from its context and into a new method. A call to the new method is inlined where the fragment was before. It is used to break code into logical units, with names that explain their purpose. 3

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

5. **profiling** is to run a computer program through a profiler/with a profiler attached. A profiler is a program for analyzing performance within an application. It is used to analyze memory consumption, processing time and frequency of procedure calls and such. 9

6. **software obfuscation** makes source code harder to read and analyze, while preserving its semantics. 1

7. **xUnit framework** An xUnit framework is a framework for writing unit tests for a computer program. It follows the patterns known from the JUnit framework for Java [Fow]. 13

Bibliography

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

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

[CK94] S.R. Chidamber and C.F. Kemerer. “A Metrics Suite for Object Oriented Design.” In: *IEEE Transactions on Software Engineering* 20.6 (June 1994), pp. 476–493. ISSN: 0098-5589. DOI: 10.1109/32.295895.

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

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

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

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

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

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

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

[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.

[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.