UiO : Department of Informatics
University of Oslo

Refactoring

An essay

Erlend Kristiansen
master thesis spring 2013

Abstract

Empty document.

Contents

List of Figures

List of Tables

Preface

Todo list

Part I

Introduction

Chapter 1

Refactoring in general

1.1 What is refactoring?

This question is best answered dividing the answer into two parts. First defining the concept of a refactoring, then discuss what the dicipline of refactoring is all about. And to make it clear already from the beginning: The dicussions in this report must be seen in the context of object oriented programming languages. It may be obvious, but much of the material will not make much sense otherwise, although some of the techniques may be applicable to sequential languages, then possibly in other forms.

sequential?

1.1.1 Defining refactoring

Martin Fowler, in his masterpiece on refactoring [2], defines a refactoring like this:

> *Refactoring* (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour. [2]

what does he mean by internal?

This definition gives additional meaning to the word *refactoring*, beyond its original meaning. Fowler is mixing the *motivation* behind refactoring into his definition. Instead it could be made clean, only considering the mechanical and behavioural aspects of refactoring. That is to factor the program again, putting it together in a different way than before, while preserving the behaviour of the program. An alternative definition could then be:

original?

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

So a refactoring primarily changes how the *code* of a program is percepted by the *programmer*, and not the behaviour experienced by any user of the program. Although the logical meaning is preserved, such changes could potentially alter the program's behaviour 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 such a thing as *software obfuscation* is to refactor. If we where to define it as a refactoring, it could be defined as a composite refactoring (see 1.5), consisting of, for instance, a series of rename refactorings. (But it could of course be much more complex, and the mechanics of it would not exactly be carved in stone.) To perform some serious obfuscation one would also take advantage of techniques not found among established refactorings, such as removing whitespace. This might not even generate a different syntax tree for languages not sensitive to whitespace, placing it in the gray area of what transformations is to be considered refactorings.

Finally, to *refactor* is (quoting Martin Fowler)

> ...to restructure software by applying a series of refactorings without changing its observable behaviour. [2]

1.1.2 Motivation

To get a grasp of what refactoring is all about, we can answer this question: *Why do people refactor?* Possible answers could include: "To remove duplication" or "to break up long methods". Practitioners of the art of Design Patterns [3] could say that they do it to introduce a long-needed pattern to their program's design. So it's safe to say that peoples' intentions are to make their programs *better* in some sense. But what aspects of the 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 those up or down in their hierarchies. Making template methods for overlapping algorithms and so on. It's all about gathering what belongs together and putting it all in one place. And the result? The code is easier to maintain. When removing the implicit coupling between the code snippets, the location of a bug is limited to only one place, and new functionality need only to be added this one place, instead of a number of places people might not even remember.

better?: functionality

The same people find out that their program contains a lot of long and hard-to-grasp methods. Then what do they do? They begin dividing their methods into smaller ones, using the *Extract Method* refactoring [2]. Then they may discover something about their program that they weren't aware of before; revealing bugs they didn't know about or couldn't find due to the complex structure of their program. Making the methods smaller and giving good names to the new ones clarifies the algorithms and enhances the *understandability* of the program. This makes simple refactoring an excellent method for exploring unknown program code, or code that you had forgotten that you wrote!

Proof?

The word *simple* came up in the last section. In fact, most basic refactorings are simple. The true power of them are revealed first when they are combined into larger — higher level — refactorings, called *composite refactorings* (see 1.5). Often the goal of such a serie of refactorings is a design pattern. Thus the *design* can be evolved throughout the lifetime

of a program, opposed to designing up-front. It's all about being structured and taking small steps to improve the design.

Many refactorings are aimed at lowering the coupling between different classes and different layers of logic. Say for instance that the coupling between the user interface and the business logic of a program is lowered. Then the business logic of the program could much easier be the target of automated tests, increasing the productivity in the software development process. It would also be much easier to distribute the different parts of the program if they were decoupled.

Another effect of refactoring is that with the increased separation of concerns coming out of many refactorings, the *performance* is improved. When profiling programs, the problem 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.

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. With the first as an obvious advantage, but with the following two being also very important in 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. So, refactoring can also add to the monetary value of a business, by increased productivity of the develompment process in the long run. Where this last point also should open the eyes of some nearsighted managers who seldom see beyond the next milestone.

1.2 Classification of refactorings

1.2.1 Structural refactorings

Basic refactorings

Extract Method
When: You have a code fragment that can be grouped together.
How: Turn the fragment into a method whose name explains the purpose of the method.

Inline Method
When: A method's body is just as clear as its name.
How: Put the method's body into the body of its callers and remove the method.

Inline Temp
When: You have a temp that is assigned to once with a simple expression, and the temp is getting in the way of other refactorings.
How: Replace all references to that temp with the expression

Move Method

When: A method is, or will be, using or used by more features of another class than the class on which it is defined.
How: Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether.

Move Field
When: A field is, or will be, used by another class more than the class on which it is defined
How: Create a new field in the target class, and change all its users.

Replace Magic Number with Symbolic Constant
When: You have a literal number with a particular meaning.
How: Create a constant, name it after the meaning, and replace the number with it.

Encapsulate Field
When: There is a public field.
How: Make it private and provide accessors.

Replace Type Code with Class
When: A class has a numeric type code that does not affect its behaviour.
How: Replace the number with a new class.

Replace Type Code with Subclasses
When: You have an immutable type code that affects the behaviour of a class.
How: Replace the type code with subclasses.

Replace Type Code with State/Strategy
When: You have a type code that affects the behaviour of a class, but you cannot use subclassing.
How: Replace the type code with a state object.

Consolidate Duplicate Conditional Fragments
When: The same fragment of code is in all branches of a condtional expression.
How: Move it outside of the expression.

Remove Control Flag
When: You have a variable that is acting as a control flag fro a series of boolean expressions.
How: Use a break or return instead.

Replace Nested Conditional with Guard Clauses
When: A method has conditional behaviour that does not make clear the normal path of execution.
How: Use guard clauses for all special cases.

Introduce Null Object
When: You have repeated chacks for a null value.
How: Replace the null value with a null object.

Introduce Assertion

When: A section of code assumes something about the state of the program.
How: Make the assumption explicit with an assertion.

Rename Method
When: The name of a method does not reveal its purpose.
How: Change the name of the method

Add Parameter
When: A method needs more information from its caller.
How: Add a parameter for an object that can pass on this information.

Remove Parameter
When: A parameter is no longer used by the method body.
How: Remove it.

Preserve Whole Object
When: You are getting several values from an object and passing these values as parameters in a method call.
How: Send the whole object instead.

Remove Setting Method
When: A field should be set at creation time and never altered.
How: Remove any setting method for that field.

Hide Method
When: A method is not used by any other class.
How: Make the method private.

Replace Constructor with Factory Method
When: You vant to do more than simple construction when you create an object
How: Replace the consgtructor with a factory method.

Pull Up Field
When: Two subclasses have the same field.
How: MOve the field to the superclass.

Pull Up Method
When: You have methods with identical results on subclasses.
How: Move them to the superclass.

Push Down Method
When: Behaviour on a superclass is relevant only for some of its subclasses.
How: Move it to those subclasses.

Push Down Field
When: A field is used only by some subclasses.
How: Move the field to those subclasses

Extract Interface
When: Several clients use the same subset of a class's interface, or two classes have part of their interfac l es in common.
How: Extract the subset into an interface.

Replace Inheritance with Delegation
When: A subclass uses only part of a superclasses interface or does not want to inherit data.
How: Create a field for the superclass, adjust methods to delegate to the superclass, and remove the subclassing.

Replace Delegation with Inheritance
When: You're using delegation and are often writing many simple delegations for the entire interface
How: Make the delegating class a subclass of the delegate.

Composite refactorings

Extract Class
When: You have one class doing work that should be done by two
How: Create a new class and move the relevant fields and methods from the old class into the new class.

Inline Class
When: A class isn't doing very much.
How: Move all its features into another class and delete it.

Hide Delegate
When: A client is calling a delegate class of an object.
How: Create Methods on the server to hide the delegate.

Remove Middle Man
When: A class is doing to much simple delegation.
How: Get the client to call the delegate directly.

Replace Data Value with Object
When: You have a data item that needs additional data or behaviour.
How: Turn the data item into an object.

Change Value to Reference
When: You have a class with many equal instances that you want to replace with a single object.
How: Turn the object into a reference object.

Encapsulate Collection
When: A method returns a collection
How: Make it return a read-only vew and provide add/remove methods.

Replace Subclass with Fields
When: You have subclasses that vary only in methods that return constant data.
How: Change the methods to superclass fields and eliminate the subclasses.

Decompose Conditional
When: You have a complicated conditional (if-then-else) statement.
How: Extract methods from the condition, then part, an else part.

Consolidate Conditional Expression
When: You have a sequence of conditional tests with the same result.
How: Combine them into a single conditional expression and extract it.

Replace Conditional with Polymorphism
When: You have a conditional that chooses different behaviour depending on the type of an object.
How: Move each leg of the conditional to an overriding method in a subclass. Make the original method abstract.

Replace Parameter with Method
When: An object invokes a method, then passes the result as a parameter for a method. The receiver can also invoke this method.
How: Remove the parameter and let the receiver invoke the method.

Introduce Parameter Object
When: You have a group of parameters that naturally go together.
How: Replace them with an object.

Extract Subclass
When: A class has features that are used only in some instances.
How: Create a subclass for that subset of features.

Extract Superclass
When: You have two classes with similar features.
How: Create a superclass and move the common features to the superclass.

Collapse Hierarchy
When: A superclass and subclass are not very different.
How: Merge them together.

Form Template Method
When: You have two methods in subclasses that perform similar steps in the same order, yet the steps are different.
How: Get the steps into methods with the same signature, so that the original methods become the same. Then you can pull them up.

1.2.2 Functional refactorings

Substitute Algorithm
When: You want to replace an algorithm with one that is clearer.
How: Replace the body of the method with the new algorithm.

1.3 The impact on software quality

1.3.1 What is meant by 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 mean 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.)

1.3.2 The impact on performance

> Refactoring certainly will make software go more slowly, but it also makes the software more amenable to performance tuning. [2]

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 [1] disproves this view in the case of polymorphism. He is doing 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.)

But is the result better?

> 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. [2]

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.

1.4 Correctness of refactorings

1.5 Composite refactorings

1.6 Software metrics

Bibliography

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

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

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