UiO : Department of Informatics
University of Oslo

Refactoring

An unfinished essay

Erlend Kristiansen
Master's Thesis Spring 2014

Abstract

Empty document.

Contents

List of Figures

List of Tables

Preface

Chapter 1

Introduction

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 discipline of refactoring is all about. And to make it clear already from the beginning: The discussions 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 behavior. [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 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:

original?

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

So 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 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 behavior. [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 series 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 development 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 behavior.
How: Replace the number with a new class.

Replace Type Code with Subclasses
When: You have an immutable type code that affects the behavior 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 behavior 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 conditional 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 behavior 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 checks 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 want to do more than simple construction when you create an object
How: Replace the constructor 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: Behavior 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 interfaces 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 behavior.
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 view 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 behavior 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

Chapter 2

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.

2.1 Design

The refactoring world of Eclipse can in general be separated into two parts: The language independent part and the the part written for a specific programming language – the language that is the target of the supported refactorings.

What about the language specific part?

2.1.1 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

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

an instance of one of its subclasses. The main responsibilities of subclasses of `Refactoring` is to implement template methods for condition checking (`checkInitialConditions`[1] and `checkFinalConditions`[2]), in addition to the `createChange`[3] method that creates and returns an instance of the `Change` class.

If the refactoring shall support that others participate in it when it is executed, the refactoring has to be a processor-based refactoring[4]. It then delegates to its given `RefactoringProcessor`[5] for condition checking and change creation.

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.

2.2 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.

2.2.1 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.

2.2.2 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 3.2.6.)

2.3 Wishful Thinking

Chapter 3

Composite Refactorings in Eclipse

3.1 A Simple Ad Hoc Model

As pointed out in chapter 2, 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.

3.2 The Extract and Move Method Refactoring

3.2.1 The Building Blocks

This is a composite refactoring, and hence is built up using several primitive refactorings. These basic building blocks are, as its name implies, the Extract Method Refactoring [2] and the Move Method Refactoring [2]. 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 which access modifier that shall be used and some not so interesting parameters.

[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`

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[1] from the Java Model 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 and delegation.

To make a whole refactoring from the processor, one have to construct a `MoveRefactoring` from it.

3.2.2 The ExtractAndMoveMethodChanger Class

The `ExtractAndMoveMethodChanger`[2] class, that is a subclass of the class `RefaktorChanger`[3], is the class responsible for composing the `ExtractMethodRefactoring` and the `MoveRefactoring`. Its constructor takes a project handle[4], the method name for the new method and a `SmartTextSelection`[5].

A `SmartTextSelection` is basically a text selection[6] object that enforces the providing of the underlying document during creation. I.e. its `getDocument`[7] method will never return `null`.

Before extracting the new method, the possible targets for the move operation is found with the help of an `ExtractAndMoveMethodPrefixesExtractor`[8]. The possible targets is computed from the prefixes that the extractor returns from its `getSafePrefixes`[9] method. The changer then choose the most suitable target by finding the most frequent occurring prefix among the safe ones. The target is the type of the first part of the prefix.

After finding a suitable target, the `ExtractAndMoveMethodChanger` first creates an `ExtractMethodRefactoring` and performs it as explained in section 2.1.1 about the execution of refactorings. Then it creates and performs the `MoveRefactoring` in the same way, based on the changes done by the Extract Method refactoring.

3.2.3 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.

[1] `org.eclipse.jdt.core.IMethod`
[2] `no.uio.ifi.refaktor.changers.ExtractAndMoveMethodChanger`
[3] `no.uio.ifi.refaktor.changers.RefaktorChanger`
[4] `org.eclipse.core.resources.IProject`
[5] `no.uio.ifi.refaktor.utils.SmartTextSelection`
[6] `org.eclipse.jface.text.ITextSelection`
[7] `no.uio.ifi.refaktor.utils.SmartTextSelection#getDocument()`
[8] `no.uio.ifi.refaktor.extractors.ExtractAndMoveMethodPrefixesExtractor`
[9] `no.uio.ifi.refaktor.extractors.ExtractAndMoveMethodPrefixesExtractor#getSafePrefixes()`

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 [3] 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 the selection. An unfix is a name that is assigned to within the 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.

Computing Safe Prefixes

A safe prefix is a prefix that does not enclose an unfix. 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` and can be fetched calling the `getSafePrefixes` method of the `ExtractAndMoveMethodPrefixesExtractor`.

3.2.4 The Prefix Class

?

3.2.5 The PrefixSet Class

3.2.6 Hacking the Refactoring Undo History

Where to put this section?

As an attempt to make multiple subsequent changes to the workspace appear as a single action (i.e. make the undo changes appear as such), I tried to alter the undo changes[9] in the history of the refactorings.

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

My first impulse was to remove the, in this case, last two undo changes from the undo manager[1] for the Eclipse refactorings, and then add them to a composite change[2] 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 [3] the undo manager, to intercept and collect the undo changes before delegating to the `addUndo` method[3] of the manager. Instead of giving it the intended undo change, a null change could be given to prevent it from making any changes if run. Then one could let the collected undo changes form a composite change to be added to the manager.

There is a technical challenge with this approach, and it relates to the undo manager, and the concrete implementation UndoManager2[4]. 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[5]. One could imagine that it might be possible to trick the undo manager into believing that you are doing a real change, by executing a refactoring that is returning a kind of null change that is returning our composite change of undo refactorings when it is performed.

Apart from the technical problems with this solution, there is a functional problem: If it all had worked out as planned, this would leave the undo history in a dirty state, with multiple empty undo operations corresponding to each of the sequentially executed refactoring operations, followed by a composite undo change corresponding to an empty change of the workspace for rounding of our composite refactoring. The solution to this particular problem could be to intercept the registration of the intermediate changes in the undo manager, and only register the last empty change.

Unfortunately, not everything works as desired with this solution. The grouping of the undo changes into the composite change does not make the undo operation appear as an atomic operation. The undo operation is still split up into separate undo actions, corresponding to the change done by its originating refactoring. And in addition, the undo actions has to be performed separate in all the editors involved. This makes it no solution at all, but a step toward something worse.

There might be a solution to this problem, but it remains to be found. The design of the refactoring undo management is partly to be blamed for this, as it it is to complex to be easily manipulated.

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

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

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

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

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

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.

Todo list