Refactoring (in) Eclipse

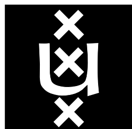
J. van den Bos

Master's thesis

August 15, 2008

Master Software Engineering

Universiteit van Amsterdam



Thesis Supervisor: Prof. Dr. P. Klint

Internship Supervisor: Drs. H.J.S. Basten

Institute: Centrum voor Wiskunde en Informatica

Availability: public domain

Contents

Abstract **5**

Preface **7**

1 Introduction **9**

1.1 Refactoring . . . 9
1.2 Refactoring Tools . . . 9
1.3 Implementing Refactoring Tools . . . 10
1.4 Refactoring in the Eclipse JDT . . . 10
1.5 Research Question . . . 10
1.6 Organization of this thesis . . . 11

2 Eclipse JDT Refactoring Development **13**

2.1 Application Architecture . . . 13
2.2 Implementation Tasks . . . 16

3 Refactoring Development Analysis **19**

3.1 Implementation Analysis . . . 19
3.2 Discovered Development Issues . . . 21

4 Proposed Development Changes **23**

4.1 Requirements . . . 23
4.2 Functionality and implementation . . . 23
4.3 Expected improvements and shortcomings . . . 26

5 Change Implementation Results **29**

5.1 Change implementation . . . 29
5.2 Change results and analysis . . . 30

6 **Summary, Conclusion and Future Work** **35**
6.1 Summary 35
6.2 Conclusion 36
6.3 Future work 37

Bibliography **39**

A **Eclipse JDT Refactoring Architecture** **41**
A.1 Introduction 41
A.2 Module View 42
A.3 Component-and-Connector View 44
A.4 Allocation View 48

B **Development Analysis Results** **51**

C **Refactoring Prototype** **53**

Abstract

Refactoring is a key activity in software maintenance and evolution. Refactoring tools exist for the most popular programming languages, but since these tools must perform extensive analysis and transformation tasks based on often complex language grammars, they are difficult to develop. As a result, refactoring tools for languages with a smaller following, such as scripting or domain-specific languages, are practically non-existent.

The refactoring functionality of the Eclipse Java Development Tools (JDT), a popular open source integrated development environment (IDE) has been examined in order to find opportunities to simplify the development of refactorings for it, with the objective of simplifying development of all refactoring tools.

Four refactoring implementations have been selected and the implementations of their initial refactoring step, checking preconditions to determine whether a refactoring can be executed, have been analyzed. This resulted in the discovery of two types of problems that complicate development and occur in all four implementations: application flow issues and semantic clones. Application flow issues refers to code that is executed either when it shouldn't be or in a different point in the normal application flow. Semantic clones refers to the same functionality being implemented multiple times.

To help refactoring developers avoid these problems, a change to how refactorings are developed is proposed. First, preconditions are made declarative and attached to refactoring implementations using the Java mechanism of annotations. Secondly, these annotations are associated with refactoring-independent and reusable precondition checking implementations. Finally, they are executed according to a fixed implementation in the refactoring base class. To validate the expected benefits, the four refactorings have been modified to function according to the proposed changes.

The original analysis has been repeated on the changed implementations, confirming the removal of the issues found in the original code. Precondition checks that are shared by multiple refactorings are reused and the standardized application flow implementation functions correctly for all four refactorings. Furthermore, the size of the source code has been measured before and after the change, showing a small increase in size, corresponding to the expected overhead of placing each precondition in a separate class. If sufficient refactorings are implemented using this approach, it is expected that reuse will reduce the total size of the source code.

It is concluded that the proposed changes simplify development of refactorings for the Eclipse JDT. Furthermore, since the approach deals with a fundamental aspect of refactoring development, a similar effect on the development of refactorings for other languages is expected.

Preface

Developing software that works correctly, does what its users want and accomodates inevitable changes with ease is exceptionally difficult. Looking at an existing system and pointing out all kinds of problems with it is much easier. Regardless of my analysis of the implementation of the Eclipse JDT refactoring tools, I am very impressed by its architecture and capabilities. I hope this thesis can contribute to its continuing success.

I would like to thank Paul Klint for offering me the opportunity to work on this project at the CWI as well as for the valuable feedback and supervision. Additionally, I would like to thank Bas Basten for his valuable feedback and supervision. Furthermore, I am indebted to all members of the SEN1 group at the CWI and everybody involved in the Master Software Engineering program at the Universiteit van Amsterdam for their enthusiasm in discussing with me and teaching me about software engineering, conducting research and writing. I have learned a great deal this year and have thoroughly enjoyed it.

I want to thank my parents, Piet and Tineke, for buying me my first computer and for their unwavering support, even when I decided not to complete my studies all those years ago. I also want to thank my wife, Nicky, who encouraged me to complete them now and did everything to support me. She makes everything worthwhile.

Jeroen van den Bos
's-Gravenhage, August 15 2008

Chapter 1

Introduction

This introduction discusses the background of developing automated refactoring tools which forms the motivation for this research. It also contains the research question along with a description of the remaining chapters of this thesis.

1.1 Refactoring

In his book *Refactoring: Improving the Design of Existing Code*, Martin Fowler provides the following definition: "Refactoring is a change made to the internal structure of a software component to make it easier to understand and cheaper to modify, without changing the observable behavior of that software component [Fowler et al., 1999]." This definition is attractive because it not only defines what refactoring is, but also why refactorings are performed. The term itself is most likely derived from the use of factoring in the following remark by Peter Deutsch: "We believe that interface design and functional factoring constitute the key intellectual content of software and are far more difficult to create or recreate than code [Deutsch, 1989]."

Refactorings are performed to improve the functional factorings of a software system in order to increase understandability and modifiability. Understandability is important because understanding software takes up a significant part of software projects: approximately 40-60% of maintenance, which in turn typically takes up two-thirds of the entire project [Pigoski, 1996]. Modifiability's importance can be explained using the first two Laws of Software Evolution [Lehman, 1996], which state that software must be continually adapted to prevent it from becoming unusable and that changing software deteriorates its structure, unless specific action is undertaken to prevent it.

1.2 Refactoring Tools

Manual refactorings are often error-prone and time-consuming [Roberts, 1999]. For instance, renaming a method requires checking that the method's new name is not yet in use as well as updating all invocations. Besides being obviously time-consuming this operation is also error-prone because polymorphism may cause a forgotten update to compile correctly but change the software's behavior inadvertently. This requires the maintainer to manually locate the changed functionality and update

the omitted invocation. As a result, these refactorings are often not performed and the software's structure deteriorates as a result of functional changes.

Automated refactoring tools can reduce these problems. If performed by a tool that can guarantee that the refactorings it performs are behavior-preserving, the error-prone aspect is mitigated. Furthermore, performing the required analysis automatically can drastically reduce the time required to perform refactorings. Automated refactoring is made possible through the use of preconditions that if satisfied guarantee that a refactoring is behavior-preserving [Opdyke, 1992]. Also, composing larger refactorings from smaller ones means that a relatively small set of automated refactorings can be used to perform a large set of refactorings [Tokuda and Batory, 2001].

1.3 Implementing Refactoring Tools

When performed by a tool, refactoring consists of at least two steps [Roberts, 1999]. The first is analysis, where the program to be refactored is analyzed in order to determine whether the desired refactoring's preconditions are satisfied. If this is the case, the second step is executed, the actual transformation of the program source code. Both steps must take both the syntax and semantics of the programming language the tool supports into consideration, making it a considerable effort to implement a refactoring tool from scratch. This may explain why refactoring tools are often integrated with other development tools such as IDEs, since these typically expose a large part of this required functionality which can be reused by the refactoring tool.

No matter how complex a refactoring tool is, in theory it only needs to be implemented once for each programming language where automated refactoring functionality is desired and then evolved along with the language it supports. Given the large developer communities and companies backing the most popular programming languages, several refactoring tools are readily available[1]. The story is different however for domain-specific languages (DSLs). Given that programs developed in a DSL must usually be maintained in the same way as a program in any other language, the smaller size of the community and availability of supporting tools such as IDEs makes the odds of having access to a refactoring tool much smaller.

1.4 Refactoring in the Eclipse JDT

In order to increase the availability of refactoring tools for DSLs it is worthwhile to investigate whether refactoring tool implementation can be simplified in order to be able to develop refactoring tools (for DSLs) faster or even generate them. To this end, it is useful to examine the implementation of existing refactoring tools in order to gain insight into the complexities of their development. Eclipse[2] is a suitable candidate for such an examination, since it is a very widely used IDE and it supports a large set of refactorings out-of-the-box, especially in the JDT, the Java Development Tooling plug-in.

1.5 Research Question

The following research question defines the activities resulting in this thesis.

[1] See http://www.refactoring.com/tools.html for an overview.

[2] See http://www.eclipse.org/.

How can development of refactorings for the Eclipse JDT be simplified?

To answer this question, the following subquestions have been formulated:

- How are automated refactorings currently developed for the Eclipse JDT?
- What parts of this task are currently complex and can benefit from improvement?
- What can be changed to reduce this complexity?

1.6 Organization of this thesis

The following chapters (2-5) answer the subquestions defined as part of the research question of this thesis. Chapter 2 describes the architecture of refactoring in the Eclipse JDT, summarizing the architecture documentation in appendix A. Chapter 3 describes the analysis and results of the investigation determining the complexity of the development work, of which the complete results are available in appendix B. Chapter 4 proposes an improvement to the problems observed in the analysis. A working prototype incorporating the proposed changes is included in appendix C. Chapter 5 discusses the implementation of the proposed changes and validates their contribution. Chapter 6 contains the conclusion.

Chapter 2

Eclipse JDT Refactoring Development

This chapter describes how automated refactorings must currently be developed for the Eclipse JDT. It is split up in two parts: application architecture and implementation tasks. The architecture section provides a context by describing the runtime flow based on the most important components. The implementation section describes the concrete steps a developer must take to develop an automated refactoring.

Since no comprehensive documentation on this subject is currently available apart from a single tutorial [Widmer, 2008], the architecture described here has been reconstructed. The result of this reconstruction is the architecture documentation included in appendix A. This chapter summarizes the parts that are relevant for this thesis.

2.1 Application Architecture

This section describes the application architecture of the refactoring component of the Eclipse JDT. It has been divided into general architecture and optional extensions, the latter describing additional functionality that does not have to be used or implemented, but is made available by the Eclipse JDT to enhance refactoring implementations.

2.1.1 General Architecture

Figure 2.1 is a UML Communication diagram showing the execution of a refactoring in the Eclipse JDT. In this case, the actual refactoring is the *Move Instance Method* refactoring, but nearly all refactorings are executed in the same manner at this level of abstraction. The grey boxes indicate objects whose classes have been implemented by a developer responsible for this specific refactoring. Following is a description of the nine steps shown in the diagram.

1: selectionChanged(). Whenever a user changes a selection in the editor or any other source code view in the Eclipse JDT environment, all registered handlers are executed, which includes all context-sensitive refactorings. This allows a refactoring's *Action* object to enable or disable itself based on the current selection. It is essential that only trivial checks are performed here, to prevent the user interface from slowing down significantly.

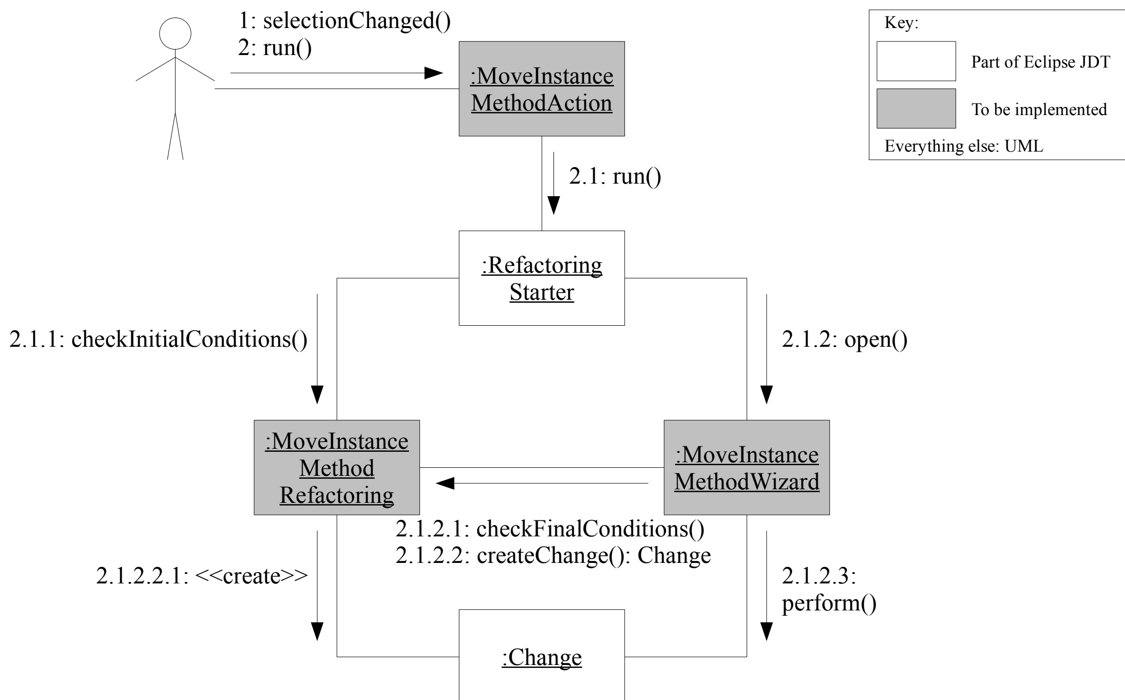


Figure 2.1: Communication diagram showing the execution of a refactoring in the Eclipse JDT.

2: run() and **2.1: run()**. When the user activates an available refactoring, the *run* method on the associated *Action* object is executed, which delegates the actual instantiation and execution to the *RefactoringStarter* object.

2.1.1: checkInitialConditions(). After the refactoring object is instantiated, the first interface method is executed, which performs the refactoring's precondition checks that can be done based on the user's selection without additional user input (in the case of this refactoring, an example is checking whether the method is not a constructor, since a constructor cannot be moved to another class).

2.1.2: open(). If the initial precondition checks are successful, the refactoring's own wizard object is activated to retrieve additional input from the user. In the case of the *Move Instance Method* refactoring, this includes information such as to which class the method is to be moved. Every refactoring uses a wizard, even if no additional user input is required. In the latter case a default implementation is instantiated that shows a preview of the proposed changes and provides progress and error reporting for the refactoring.

2.1.2.1: checkFinalConditions(). After the user has entered additional information, the second interface method is executed, which performs the refactoring's precondition checks that take the entered information into account (in the case of this refactoring, an example is checking whether the selected target class does not already contain a method with the same signature).

2.1.2.2: createChange(): Change and **2.1.2.2.1: <<create>>**. If the final precondition checks are also successful, the third interface method is executed, creating a *Change* object encapsulating all changes to be performed on the workspace in order to finalize the refactoring. Storing all changes in a single object allows previewing them and supports atomic undo/redo operations.

2.1.2.3: perform(). If the user agrees with the preview shown by the wizard, continuing will commit the changes in the *Change* object to the workspace, concluding the refactoring execution.

2.1.2 Optional Extensions

The architecture discussed in the previous section is the minimum for refactorings to adhere to. Two significant extensions are available: scripting support and processor-based refactorings. These extensions are described in the following sections.

Scriptable Refactorings

There are two typical scenarios where scriptable, or non-interactive, refactorings are useful. The first is for serializing performed refactorings, for instance to perform previously applied refactorings again on a previous version of a code base without having to remember and enter the user settings again. The second scenario is for composing refactorings: large and complex refactorings (such as introducing design patterns) can be composed from smaller refactorings. If those refactorings can be strung together using each other's input and output, the larger refactorings can be treated as atomic operations, relieving the user from having to manually apply the separate steps.

A scriptable refactoring must implement the *IScriptableRefactoring* interface, or more commonly, derive from a partial implementation in the *ScriptableRefactoring* abstract base class. If referenced from a script or other non-interactive part of the Eclipse environment (such as the Refactoring History Manager), the name of the refactoring is mapped to a *RefactoringContribution*-derived class, which is then instantiated and passed the arguments from the script. This contribution object is then requested to create a *RefactoringDescriptor*-derived class, which it passes the validated arguments. This descriptor then finally instantiates the appropriate refactoring object using *IScriptableRefactoring*'s *initialize* method. The resulting refactoring object is ready to be executed without requiring additional user input.

Processor-based Refactorings

Eclipse is an integrated development environment (IDE) offering a large set of development tools in an integrated manner to the software engineer using it. As a result, a refactoring that changes several source files may have impact on some of the other integrated tools. For example, if a breakpoint for use by the integrated debugger is placed on a line of code that is moved to another source file by a refactoring, the breakpoint should be moved as well. However, it is outside of the scope of a refactoring's implementation to deal with each of these concerns itself (besides seriously limiting the extensibility of the platform if each change should lead to modifications in every related piece of functionality). To achieve this, processor-based refactorings exist.

A processor-based refactoring is basically a standard refactoring, with the exception that between each call to a method of its interface, a list of previously configured participants is called in order to keep them up to date of the impending or executed changes. In the example used before, this can be a BreakpointMoveParticipant, that is called after the refactoring checks its preconditions to check whether the breakpoint can be moved. After the refactoring's changes have been executed, the participant is then called again in order to execute its specific changes. The execution of the processor and its participants by a controlling *RefactoringProcessor* allows relevant state to be shared (e.g. the participant may examine the changes executed by the refactoring in order to make its own adjustments).

2.2 Implementation Tasks

Implementing an automated refactoring consists of three main activities: implementing the refactoring's functionality itself (both analysis and transformation), implementing the hook that allows the refactoring to be accessible to the user and implementing the user interface to allow the user to enter information required to execute the refactoring. This final step is optional, since some refactorings do not require user input apart from activation. The following sections discuss these three tasks.

2.2.1 Refactoring functionality

To implement a refactoring, a *Refactoring*-derived class must be implemented. As discussed in the General Architecture section, this consists of implementing the two refactoring steps, divided over three methods. The analysis step is split up over the *checkInitialConditions* and *checkFinalConditions* methods. The transformation step is implemented by the *createChange* method.

Except for a progress monitor, no arguments are passed in to checkInitialConditions, which must verify that the user selection is compatible with this refactoring. A refactoring object is instantiated using the current user selection, so the check can be performed on its own fields. The *checkFinalConditions* method works similarly, as only a progress monitor is passed in as well. This method however must check whether the entered user information (either in the accompanying wizard or arguments in a script) is compatible with this refactoring in the current situation. These values are also stored in the object itself and accessed by the wizard or script through setters. Both must return a *RefactoringStatus* object that signals success if it does not contain a fatal error.

The second refactoring step, source code transformation, is encapsulated by the single and final method call, *createChange*. Similarly, only a progress monitor is passed to it, but this time a *Change* instance must be returned. A *Change* instance is an object containing all changes that must be performed to the workspace in order to complete the refactoring. Since the conditions for this refactoring have already been checked, no explicit status is returned, although exceptions may be thrown to signal that the change cannot be created due to unexpected problems.

2.2.2 Selection and activation

Menu commands, keyboard shortcuts and context menus are all represented by instances of *Action* subclasses in Eclipse. Since a refactoring's applicability is related to the current selection, a refactoring is typically connected to the user interface through a *SelectionDispatchAction*-derived class. To implement such a class, two methods must be implemented: *selectionChanged* and *run*.

The *selectionChanged* method is executed each time the menu that the object is associated with is about to be shown, allowing a just-in-time decision for each refactoring to decide on their availability. This decision is made based on the current selection, which is passed in as an argument. A selection object typically contains a reference to the file that is currently open and the begin and length of the selection in it. Based on this information, a quick analysis may be performed (to prevent the user interface from slowing down) and the *Action*'s base class *setEnabled* method called to finalize the outcome.

When an action is visible and activated by the user, its *run* method is executed, again passing in the current selection as an argument. This selection is then used to instantiate the actual refactoring class and the refactoring is started using the *RefactoringStarter* helper class.

2.2.3 User interface and customization

If additional user input is required after the refactoring has been activated (such as in the *Move Instance Method* example, this would be choosing a class to move the selected method to) a wizard may be implemented to facilitate the interaction. To create a wizard, at least one *RefactoringWizard*-derived class must be implemented, along with one or more *RefactoringWizardPage*-derived classes. The former encapsulates the wizard itself (including some default pages) and the latter encapsulate the individual pages the wizard consists of. No actual graphical user interface (GUI) code has to be developed, because this has all been encapsulated to make sure the overall Eclipse GUI is consistent. The wizard code consists solely of specifying the functionality the wizard should expose.

The main wizard class is little more than a title and defining which pages should be shown, along with their order of appearance. The pages themselves consist of defining the types of information that should be requested from the user (such as a string or a value from a list), along with the name of the variable and code to validate a correct entry. The individual pages are responsible for calling the appropriate setters in their associated refactoring object once the user advances past them. Every wizard inherits a standard preview page as well as a final page with a progress bar.

Chapter 3

Refactoring Development Analysis

This chapter describes issues related to the development of refactorings for the Eclipse JDT, that complicate the implementation tasks and are therefore candidates for improvement in order to simplify the overall development work. It is split up in two parts: a description of the analysis process that has led to the discovery of these issues and a description of all discovered types of issues. Each issue type includes several examples for illustration purposes. A full list of all instances found of each type is included in appendix B.

3.1 Implementation Analysis

The following steps make up the analysis performed that have led to the discovery of the issues detailed in the next section:

3.1.1 Selection of Relevant Classes

The Eclipse JDT contains 43 refactorings, where a refactoring is defined as a non-abstract implementation of the Refactoring or RefactoringProcessor base class that is accessible through the Eclipse user interface in order to perform a distinct refactoring. Of these refactorings, four refactorings were selected based on the following criteria:

- The set includes the *Introduce Indirection* refactoring, because it is the subject of the refactoring tutorial [Widmer, 2008] which may be a good reference for analyzing the implementation.
- The refactorings all operate on the same language construct, so that issues related to reuse can possibly be discovered.
- The set includes refactorings that implement the *IScriptableRefactoring* interface as well as the *ProcessorBasedRefactoring* base class in order to cover all types.

Table 3.1 shows the four selected refactorings and their relation to the criteria used to select them. The selected set meets the selection criteria: the set includes the *Introduce Indirection* refactoring, all refactorings operate on the *method* language construct, all refactorings are scriptable (the Eclipse JDT only includes refactorings that are scriptable) and two out of four selected refactorings are processor-based.

Refactoring	Construct	Scriptable	Processor-based
Introduce Indirection	Method	Yes	No
Extract Method	Block, Method	Yes	No
Move Instance Method	Class, Method	Yes	Yes
Rename Virtual Method	Method	Yes	Yes

Table 3.1: Selected refactorings related to the selection criteria.

3.1.2 Selection of Relevant Methods

As described earlier, tool-based refactorings consists of (at least) two distinct steps: precondition checking and source code transformation. As detailed in the application architecture documentation, precondition checking takes place in three methods of each refactoring: the *SelectionDispatchAction*-derived class' *selectionChanged* method and the *Refactoring*-derived (or *RefactoringProcessor*-derived) class' *checkInitialConditions* and *checkFinalConditions* methods. Source code transformation takes place in the *Refactoring*-derived (or *RefactoringProcessor*-derived) class' *createChange* method.

The methods investigated in this research are those related to initial precondition checking, specifically, the *selectionChanged* and the *checkInitialConditions* methods. Both these methods deal with the initial stage of a refactoring, namely checking whether a refactoring can be executed based solely on a user's selection in the user interface. Reason for investigating this part of the refactorings is that it is expected that the potential for reuse will be greatest here, since once a refactoring has started, each successive step will become increasingly specific, making it harder to find issues that may be improved generally. Based on the fact that 43 refactorings already exist in the Eclipse JDT, harnessing reuse is a viable strategy to simplify development by increasing productivity and quality [Mili et al., 1995].

3.1.3 Method Analysis

Opdyke's original catalogue of refactorings shows that refactorings often have multiple preconditions [Opdyke, 1992]. However, the interface defined by the *SelectionDispatchAction* and *Refactoring* base classes only define a single method to encapsulate all these preconditions. In order to analyse the functionality of these methods, each method has been split up into separate preconditions. A precondition for this purpose is defined as a group of related statements that for some inputs generate a fatal error status, with the goal of terminating the execution of the refactoring. The contents of methods that are called is also taken into account, up to the boundary of the refactoring components (because outside these bounds the code is not considered to be specific to precondition checking).

Furthermore, the general application flow has been examined by stepping through each refactoring in different scenarios. For each precondition check, a scenario has been executed where that precondition check is expected to fail, as well as a general scenario where each precondition check is expected to succeed.

3.2 Discovered Development Issues

The following sections each describe a type of issue that was discovered during the analysis of the refactoring implementations. Each section contains a description of the issue, why it is considered a problem and some examples to illustrate it. Appendix B contains a full list of all instances of each issue discussed here for reference.

3.2.1 Application Flow

An application flow issue refers to code that is executed either when it shouldn't be at all or at a different point in the normal application flow. These defects are problematic because they may cause failures. In the event that they do not cause failures, they may still be harmful because they reduce the maintainability of the system (confusing maintainers) and may have negative effects on system qualities such as performance (for example when code is run multiple times when it shouldn't be). In summary, application flow issues are related to *when* code is executed.

Example 1: Skipping precondition checks

One of the precondition checks in the *MoveInstanceMethodRefactoring*'s *checkMethodBody* method has been placed inside the body of an unrelated *if*-block, causing the check to be skipped (and as a result assumed to be successful) if the expression at the top of the block happens to be *false*. The result is that this precondition is sometimes checked and sometimes skipped while the reason is unrelated to the check's applicability to the selection at hand.

Example 2: Executing unrelated code

Both the *IntroduceIndirectionRefactoring* and *ExtractMethodRefactoring* execute code unrelated to precondition checking in their *checkInitialConditions* method. This code could just as easily be moved to either the constructor or to one of the other methods executed and the best location is most likely close to where the state that is initialized is actually used. This issue is a result of the localized nature of the refactorings' implementation: since all precondition checking and program source transformation is executed in one and the same instance, it's tempting to add some additional code that needs to be executed to one of the methods' implementation, regardless of whether that would be the best location for it.

3.2.2 Semantic Clones

Use of the term semantic clones refers to the following definition: "A piece of code, *A*, is similar to another piece of code, *B*, if *B* subsumes the functionality of *A*; in other words, they have "similar" preconditions and postconditions. We call such a pair (*A*, *B*) a *semantic* clone [Bellon et al., 2007]." Semantic clones are a problem because they are basically the same functionality implemented multiple times, degrading maintainability considerably (for example when a single precondition check is implemented multiple times, a change to the language specification that affects the precondition check must be updated in all implementations). In summary, semantic clones issues are related to *what* code is executed.

Example 1: Multiple definitions of movable

The *Move Instance Method* refactoring has several implementations for checking the eligibility of a method to be moved. Preconditions such as the selected method not being static or a constructor are implemented in *MoveInstanceMethodProcessor*'s *checkMethodDeclaration* method and *isApplicable* method, as well as in *RefactoringAvailabilityTester*'s *isMoveMethodAvailable* method. This example clearly illustrates the problems these clones cause, since only two out of these three methods check that the method is not an annotation and only one checks that the method is not synchronized.

Example 2: Multiple definitions of extractable

When executing the *Extract Method* refactoring, a basic check is initially executed to determine whether the selected code is extractable. Later on, a more elaborate check is performed, taking into account several additional requirements (such as whether a *super* call exists in the selection). Although it is understandable that multiple implementations exist for performance reasons, the implementations do not themselves reveal whether they are optimized for speed or accuracy. If the quick, but inaccurate check is reused by a developer that believes it to be accurate, problems will arise.

Chapter 4

Proposed Development Changes

This chapter describes the proposed changes to the refactoring implementation in order to mitigate the issues described in the previous chapter. It includes a description of the requirements, functionality and implementation as well as a discussion of the expected effects of applying these changes, both positive and negative. A full technical description in the form of source code to a working prototype is included in appendix C, that has not been simplified and can be used as an implementation guide to perform the changes on the existing code.

4.1 Requirements

To resolve the issues identified in the previous chapter, any proposed change should have the following effects on the development of refactorings:

- Make precondition checks independent from a specific refactoring in order to enable reuse.
- Promote reuse by giving individual precondition checks a clear name and making it easy to find which checks exist.
- Ensure that precondition checks are always executed in the same way and only at the appropriate time.

The following section describes a solution that satisfies these requirements.

4.2 Functionality and implementation

The proposed changes basically consist of three components. First, the association between precondition checks and refactorings is made declarative instead of imperative (as it currently is). Second, the precondition checks are encapsulated so that they are fully independent of a specific refactoring and reusable. Finally, the responsibility for implementing the application flow concern is removed from the refactoring developer to a standardized base class implementation that cannot be modified (for a single refactoring).

4.2.1 Imperative to declarative

In the current situation, refactoring developers do not explicitly define preconditions and their related checks. Instead they are required to implement two methods (*selectionChanged* and *checkInitialConditions*) that perform all of the initial precondition checks. Being standard Java methods, the implementation is imperative. To make refactoring implementations more readable, preconditions should be made declarative. Instead of implementing all checks directly, each precondition is named and declared as part of the refactoring, as shown in the following code:

```
@Preconditions({ RequireMethodSelection, RequireNoSyntaxErrors })
public class RenameVirtualMethodProcessor extends RefactoringProcessor {}
```

In this example, the *RenameVirtualMethodProcessor* refactoring is associated with two preconditions: *RequireMethodSelection* and *RequireNoSyntaxErrors*. The *@Preconditions* reference in the example is a Java annotation [Gosling et al., 2005]. It has a single default argument that requires a list of values from a *Precondition* enumeration that ties each precondition name to a single implementation.

4.2.2 Reusable precondition checks

Making preconditions declarative only solves half the problem: if the preconditions are still to be checked by the refactoring implementation itself, then semantic clones will probably still be developed for each specific refactoring. To resolve this, the declared preconditions are associated with single implementations that are independent of a specific refactoring, making them fully reusable. The mechanism used to connect a precondition's name to a single implementation is Java's extensive enumeration functionality. An excerpt of the *Precondition* enum is shown in the following code:

```
public enum Precondition {

    RequireMethodSelection(new RequireMethodSelectionPrecondition()),
    RequireNoSyntaxErrors(new RequireNoSyntaxErrorsPrecondition());

    private final IPrecondition _checker;

    Precondition(IPrecondition checker) {
        _checker = checker;
    }

    public IPrecondition getPreconditionChecker() {
        return _checker;
    }
}
```

In Java, enumeration values are basically instances of their enumeration class. An enumeration's declaration starts with a comma-separated list of names that make up the enumeration values. These value declarations also serve as constructor calls by allowing arguments to be passed in that match the enumeration constructor's signature.

In the definition above, each value is associated with an implementing class. As a result, a refactoring's precondition check implementations can be retrieved by reading the class' annotation, retrieving its enumeration values and calling the *getPreconditionChecker* method on each of these values.

4.2.3 Removing application flow

The next step is ensuring that the previously described method of declaring preconditions and single implementations are actually used by the refactoring (developer). In a typical maintenance situation, if confronted with choosing between both the original and the new method of precondition checking, it may be tempting to use the original method, hindering the effects of this change. This can be resolved by modifying the refactoring base class so the initial precondition checking methods can no longer be directly implemented by the refactoring developer. Furthermore, the fixed implementation will automatically perform the annotation, enumeration and implementation retrieval and execution, making this method of precondition checking the only way to check preconditions, as well as ensuring that it is always performed in the same way. The following implementation of the *checkInitialConditions* method illustrates this:

```
public abstract class AnnotatedRefactoring {

    private Object _input;
    private int _start, _length;

    protected AnnotatedRefactoring(Object input, int start, int length) {
        _input = input;
        _start = start;
        _length = length;
    }

    public final RefactoringStatus checkInitialConditions() {
        RefactoringStatus status = new RefactoringStatus();
        for (IInitialPrecondition cnd : Precondition.
            getInitialPreconditions(this)) {
            RefactoringStatus result = cnd.checkInitialCondition(_input,
                _start, _length);
            status.merge(result);
        }
        return status;
    }
}
```

The refactoring is constructed with the values required to perform precondition checking (the *input* represents the currently active object, usually in the form of a program source file, the *start* and *length* values refer to the current selection inside the active object). The *checkInitialConditions* method is marked *final*, which means it cannot be overridden by derived classes implementing concrete refactorings. When called, the method uses the static *getInitialPreconditions* method to retrieve its precondition values and their associated implementations and then proceeds to execute them, passing the current selection to each checker for analysis. The result of each check is collected and returned, ensuring its behaviour is identical to the original implementation. The following implementation of the *getInitialPreconditions* method illustrates the checker implementation retrieval:

```
public enum Precondition {

    public static IInitialPrecondition[] getInitialPreconditions(
        AnnotatedRefactoring refactoring) {
        return getPreconditions(refactoring, IInitialPrecondition.class);
```

```
    }

    @SuppressWarnings("unchecked")
    private static <T> T[] getPreconditions(AnnotatedRefactoring
        refactoring, Class<T> type) {
        Preconditions cnd = refactoring.getClass().getAnnotation(
            Preconditions.class);
        ArrayList<T> preconditions = new ArrayList<T>();
        for (Precondition value : cnd.value()) {
            if (type.isInstance(value.getPrecondition())) {
                preconditions.add((T) value.getPrecondition());
            }
        }
        return preconditions.toArray((T[]) Array.newInstance(type, 0));
    }
}
```

4.2.4 Additional optimization

The only difference between the *selectionChanged* and the *checkInitialConditions* method is the set of preconditions checks that they must execute. The *selectionChanged* method must execute only checks that are quickly executed, in order not to slow down the user interface as the user traverses a context menu. The *checkInitialConditions* method however must run all checks because the result is required to determine whether the refactoring may continue. This can be solved using polymorphism.

In the previous example the *checkInitialConditions* method only executes precondition checks that implement the *IInitialPrecondition* interface, but can easily be extended to execute precondition checks implementing the *ISelectionPrecondition* interface as well. To only perform quick checks, the *selectionChanged* method can be added that only executes precondition checks implementing the *ISelectionCondition* interface. Finally, precondition checks may choose to implement both interfaces, in order to expose both a quick and accurate version of the same check.

4.3 Expected improvements and shortcomings

Based on these changes, some improvements and shortcomings are expected in the area of refactoring development, which are described in the following sections.

4.3.1 Less application flow issues

Application flow is no longer in the hands of the refactoring developer, preventing any problems from occuring in this area. In order to implement a refactoring's initial precondition checking, the refactoring class must be annotated with relevant preconditions, including some that may have been implemented for this refactoring specifically (because none of the existing refactorings have had this precondition yet). Each individual precondition checker implements a single concern: checking whether that specific precondition is satisfied, oblivious to what actual refactoring is executing or what other preconditions are checked.

Since precondition checks are independent of refactorings, they cannot mix unrelated code such as additional transformation initialization code, forcing the developer to place this code in the constructor of the refactoring or another appropriate location.

4.3.2 Less semantic clones

Every precondition check that is implemented must be added to the *Precondition* enumeration. Using a good naming convention, attempting to add a precondition check that already exists should point to the existing declaration and implementation (since duplicate names are not allowed in enumerations). Furthermore, the independence of precondition checkers to actual refactorings and other precondition checks makes them ideally suited to be reused extensively.

4.3.3 Effects on code size

As described before: application flow code is no longer necessary and a reduction in semantic clones is expected as well. This should both lead to a reduction in total code size, simplifying maintenance and program comprehension (for new developers attempting to add a refactoring).

At the same time, some increase in code size is expected due to additional overhead caused by each precondition check being placed in its own class, as well as having to add an entry for each precondition in the *Precondition* enumeration. The expected increase caused by this is however small, since the interface each precondition check must implement consists of only a single interface method.

4.3.4 Possible naming issues

An important risk associated with this approach is that its success depends on developers taking the time to read the list of existing precondition checks in the *Precondition* enumeration before implementing any new checks. However, all reuse depends on developer's willingness to support it, although this approach does require every new check to be given a name and added to the enumeration, making it more likely for developers to come across existing checks that they can reuse.

Choosing an appropriate name for a precondition is an issue as well: a precondition can often correctly be named using different words. For example, a precondition that requires that there are no syntax errors in the current file can be named *RequireNoSyntaxErrors*, *RequireSuccessfulCompilation* or *ForbidSyntaxErrors*. The only solution to this is requiring the use of a controlled vocabulary with naming rules.

4.3.5 Implicit dependencies

Preconditions may have dependencies on other preconditions. For example, the precondition *MethodNotAConstructor* assumes that a method has been selected, since it does not make sense on any other selection. Effectively, it depends on the precondition *RequireMethodSelection*. In both the original and the changed implementation, these dependencies can be resolved using order of definition: checking for *RequireMethodSelection* first and *MethodNotAConstructor* second yields the expected behaviour.

If checks are executed in a different order, unclear errors may be reported on invalid selections. For example, in the previous example, if the *MethodNotAConstructor* precondition is checked before the *RequireMethodSelection* precondition while a method has not been selected, the *MethodNotAConstructor* precondition check will fail and report an error. This is technically correct but may be confusing to the user.

Chapter 5

Change Implementation Results

To validate that the changes proposed in the previous chapter actually resolve or reduce the effects of the issues found and contribute to simplifying the development of refactorings for the Eclipse JDT, the changes have been implemented and the resulting modified system analyzed. This chapter describes the implementation and results of implementing these changes.

5.1 Change implementation

The working prototype from appendix C has been integrated into the Eclipse JDT. The following steps have been executed for all four previously examined refactorings to modify them to adhere to the proposed change.

5.1.1 Insert modified base class

Instead of deriving from the standard *Refactoring* or *RefactoringProcessor* base class, an additional base class was inserted into the class hierarchy to provide all refactoring classes with the required annotation retrieval and precondition checker execution functionality. This class is called *AnnotatedRefactoring*. Although some classes derive from *Refactoring* and some classes derive from *RefactoringProcessor*, since their initial precondition checking interface is identical (both have the *checkInitialConditions* method with the same signature) this posed no problems.

5.1.2 Convert precondition checks

The precondition checks identified in chapter 3 were all converted to *IInitialPrecondition* or *ISelectionPrecondition* implementing objects and associated with a precondition annotation by adding it to the *Precondition* enumeration. Each precondition was then associated with its refactoring implementation by adding it to the *@Preconditions* annotation on the refactoring class. In case of a semantic clone, the existing implementation was reused and the clone was discarded.

Refactoring	Unit tests
Introduce Indirection	31
Move Instance Method	48
Rename Virtual Method	181
Extract Method	392

Table 5.1: Unit test counts for the converted refactorings.

5.1.3 Verify changed functionality

To verify that the changes did not modify the refactoring's expected behaviour, each refactoring's existing unit tests were run to determine that their behaviour was unmodified. Each unit test suite contained unit tests testing both failure and success scenarios. The total amount of unit tests available for each refactoring is shown in table 5.1.

5.2 Change results and analysis

This section describes what precondition checks have been extracted and moved to independent classes and discusses the results of the changed approach to reuse and application flow as well as the impact on code size.

5.2.1 Precondition check classes

After the changes were implemented, each refactoring was changed into an annotated class referencing several preconditions. The preconditions declared on each modified refactoring are shown in table 5.2. Because the refactorings were modified one at a time in a left-to-right order, the left-most's implementation is the one that all other refactorings referencing the same precondition have (for example, the *Rename Method* and *Extract Method* refactorings use *Move Instance Method*'s implementation of the check for the *NoSyntaxErrors* precondition.

5.2.2 Effects of increased reuse

Of the 31 precondition checks implemented, six were reused, of which three multiple times. Reuse of a precondition check occurred ten times in total. None of the semantic clones discovered during the analysis of the original implementation remain, since all have been discarded. Several discoveries were made during the implementation and reuse of the precondition checks.

First, reuse causes the precondition checks to be tested more thoroughly. For example, the *RequireMethodSelection* precondition checker originated from the *Introduce Indirection* refactoring but was reused on the *Move Instance Method* and *Rename Virtual Method* refactoring. Two test cases of the latter found two defects in this precondition check: it did not resolve native and annotation methods correctly, throwing an exception. This was not tested by the *Introduce Indirection*'s own test suite.

Refactoring (right) **Precondition (down)**	**Introduce Indirection**	**Move Instance Method**	**Rename Virtual Method**	**Extract Method**
RequireMethodSelection	•	•	•	
SelectionIsModifiable	•	•	•	•
MethodNotInAnnotation	•	•		
MethodNotInLocalClass	•			
MethodNotInAnonymousClass	•			
MethodNotAConstructor	•	•		
RequireNoSyntaxErrors		•	•	•
MethodNotStatic		•		
MethodNotAbstract		•		
MethodNotNative		•	•	
MethodNotSynchronized		•		
MethodNotInInterface		•		
MethodDoesNotReference NonLocalTypes		•		
MethodDoesNotUseSuper		•		
MethodDoesNotReference EnclosingTypes		•		
MethodNotRecursive		•		
MethodHasSourceClass ParameterOrField		•		
MethodIsVirtual			•	
RequireBlockOrInitializerSelection				•
RequireStatementSetOr SingleExpressionSelection				•
BlockNotLefthandSideOfAssignment				•
BlockDoesNotUseSuperOrThis				•
ExpressionNotTypeReference				•
ExpressionNotMethodName				•
ExpressionNotNamePartOfDeclaration				•
BlockNotBodyOfAnonymousClass				•
BlockDoesNotStartAfterDoKeyword				•
ExpressionNotFoInitializer				•
ExpressionNotForIncrementer				•
ExpressionNotSingleCaseStatement				•
BlockDoesNotContainUnselectedBranchTarget				•

Table 5.2: Implemented precondition checks and classes using them.

Secondly, availability of an easily reusable library of precondition checkers makes it easy to experiment. For example, the *NoSyntaxError* precondition is applied to all refactorings except the *Introduce Indirection* refactoring, even though there does not appear to be a reason for its omission for this refactoring. It turns out to be simple to create a scenario where the *Introduce Indirection* refactoring throws an exception because of a syntax error in the program source file the refactoring is activated on. Adding the *NoSyntaxErrors* precondition changes the refactoring to provide the user with an explanation upon executing the scenario again.

5.2.3 Effects of application flow changes

Implementing application flow is now only a part of precondition checking at the level of individual precondition checks, since an individual refactoring no longer has any influence over the order of execution. The result is that none of the duplicate execution, unneeded execution or program flow errors reported in the analysis of the original implementation remain. The implementation of these independent precondition checks has resulted in some additional observations.

First, precondition checking code is often intertwined with initialization code (that could have run before precondition checking) and input checking and program source transformation code (that could have run after precondition checking). The initial analysis suggested that these were cases of misplaced code, but in most cases it turns out that this code is also relevant for precondition checking. Most code related to precondition checking produces intermediate data structures that are required by the successive steps of the refactoring. In order to be able to move this functionality to a separate precondition check class, the code related to successive steps has been factored in public methods so that it can be called from the refactoring in a later phase of execution. This does produce a dependency from the refactoring to the precondition check class, but that is not a problem: the precondition check must be independent of individual refactorings, but the other way around is acceptable, especially compared to the alternative of duplicating the code.

Secondly, unit tests must be written to test refactoring behaviour and not implementation. This was an issue with the *Extract Method* refactoring, where the unit testing code inspected and modified the internal structure of the refactoring between precondition steps and program source transformation. Since a fully succeeding unit test suite was selected as a criterium for determining success, the implementation of independent precondition checks for the *Extract Method* refactoring was complicated. In the end, the original precondition checks were also executed alongside the annotated versions, but their output ignored. This allowed the original unit test suite to run but still test the new precondition checks.

Finally, most precondition checks are related to a specific type of selection by the user. Reuse is therefor greatest among refactorings that have the same basic selection criteria. This explains why the *Extract Method* refactoring shares only a single general precondition check with the other refactorings: it operates on sets of statements or single expressions, while the other refactorings operate on methods.

5.2.4 Effects on code size

A major effect on code size has not been observed. The cause for this has been discussed in the previous section: most of the precondition checking code sets up a significant amount of internal state in the refactoring object, but because this state is required to perform successive steps in the refactoring execution, this code has not been removed from the refactoring entirely. In several cases the relevant

Version	Introduce Indirection	MoveInstance Method	Rename VirtualMethod	Extract Method	Precondition Checks	Total
Original	926	2.026	933	1.437	0	5.322
Annotated	885	1.671	869	805	1.465	5.695

Table 5.3: Lines of code for the refactoring implementations.

code has been moved to the precondition check class and is then called from the refactoring class. In these cases however it would be misleading to not count those lines of code, as it would be similar to moving all the code to a helper class and then claiming that developing a refactoring is nothing more than calling three methods.

Table 5.3 shows the lines of code for the refactoring classes before and after implementing the changes. The first row shows the size of the original refactoring implementation. The second row shows the size of the annotated implementation. The *Precondition Checks* column shows the total size of the precondition checks that are now independent from the refactorings. The *Total* column shows the total size of the implementations.

More than half of the 7% increase in size of the changed implementation is caused by the *Extract Method* refactoring, which has 13 precondition checks that are not shared with any of the other refactorings. Reuse is not possible because these precondition checks deal specifically with block selections, while the other refactorings all deal with method selections. In the case of the other refactorings, a small amount of reuse has already compensated the increased size of having a separate class for each precondition check.

Chapter 6

Summary, Conclusion and Future Work

This chapter contains a short summary of the performed research, the conclusions drawn based on the results and a discussion of relevant future work.

6.1 Summary

This research has attempted to find an answer to the question:

How can development of refactorings for the Eclipse JDT be simplified?

In order to answer this question, the current implementation of refactorings in the Eclipse JDT has been examined and an architecture description has been developed. The architecture shows that a refactoring implementation is split up into three steps, the first concerned with analyzing the user selection and determining whether the refactoring can be executed on this selection, the second with determining whether the additionally entered information is correct and the third and final step with transforming the program source code to complete the refactoring.

The first of these steps has been examined in order to determine whether development in this step can be simplified for refactoring developers. The implementations of this step of four different refactorings have been split up into individual precondition checks to assess any issues. The examination resulted in the discovery of two important issues that severely complicate development: the developers' responsibility to integrate multiple precondition checks into single method calls causes multiple types of application flow defects and individual checks are often implemented multiple times (semantic clones).

In order to simplify development, a change has been proposed that requires each precondition check to be implemented separately from the other precondition checks as well as independent of a specific refactoring. Association between refactoring and preconditions checks is realized using Java annotations, making the association declarative. A Java enumeration is used as a central repository to link precondition checks to their implementation and the execution of the checks is delegated to a base class.

To determine the effects of this change on the actual implementation of refactorings, the examined four refactorings have been modified to function according to the proposed changes. The analysis performed on the original implementation has been repeated and shows that the application flow errors and semantic clones have been removed using this approach. Additionallly, code size has

been measured and shows that the overhead does increase code size because each precondition check now has its own class, but that increased reuse can quickly make up for this effect.

6.2 Conclusion

Determining whether something is easier than before is difficult, because the definition of easier may be different for every software engineer, including refactoring developers. It is however safe to assume that if a developer, to reach his goal, has to complete a certain amount of tasks, that his work is simplified if it can be reduced to a subset of those tasks. Providing however that the remaining tasks are not (very) different (because then it may be argued that the remaining tasks are more difficult, offsetting the decrease in complexity by having to do less work).

Part of implementing a refactoring is implementing code that checks whether all preconditions are satisfied in order to determine whether execution of the refactoring may continue. Implementing these checks consists of making a list of preconditions and implementing them one-by-one. Alternatively, existing implementations may be sought, for instance in the existing code base and copied or referenced to increase productivity.

The changes proposed in this thesis may cut down the effort in the final steps significantly: finding existing implementations has been simplified because there now exists a “master list” of refactorings in the *Precondition* enumeration. The fact that every precondition check must be named in this enumeration may even prevent accidental duplication: if a name already exists the check cannot be implemented again. Additionally, as each precondition check is implemented in isolation and executed automatically by the refactoring base class, the developer does not have to create the code that correctly runs all the checks and collects the results. There are some caveats however.

First, if a refactoring depends solely on preconditions that are unique to the refactoring, there is no reuse. Secondly, when the list grows, it may be easy to read past some preconditions and implement them again, using a different name. This can be prevented by using a controlled vocabulary that places strict rules on the naming of preconditions. However, such rules will typically be unable to prevent all confusion (and as a result, duplication).

Measures of code size before and after the change show that the overhead of placing each refactoring in its own class does increase the amount of code required if no reuse occurs. Effective reuse then is the key to whether this approach will reduce code size in the long run. Considering that the amount of existing language constructs to apply refactorings to is not very large (expressions, blocks, methods, classes, fields and a few more) along with the amount of existing refactorings in the Eclipse JDT (43 currently), it is realistic to expect considerable reuse when converting all refactorings to this approach and following that, a reduction in code size.

The proposed change then effectively prevents application flow errors by letting the refactoring developer focus entirely on implementing individual precondition checks, consists of a development method that makes it very difficult to accidentally duplicate existing code and has been shown to make up for its increase in code size with relatively limited reuse. As a result, using it will simplify refactoring development for the Eclipse JDT considerably.

Additionally, since precondition checking is a universal step in automated refactoring, this approach can simplify the development of other refactoring tools as well. The annotated refactoring classes from the Eclipse JDT can even be used as a reference for which preconditions to consider for refactoring tools targeting a language similar to Java.

6.3 Future work

The following sections discuss various possibilities for improving the results or extending the research described in this thesis.

6.3.1 Handling dependencies

As described in chapter 4, preconditions may have dependencies on other preconditions. Instead of requiring the refactoring developer to define preconditions in the correct order, the proposed changes may be extended by allowing a precondition's dependencies to be defined. The result would be that declaring *MethodNotAConstructor* on a refactoring implicitly includes *RequireMethodSelection*. The approach to this could be similar to the change proposed in this thesis: annotate preconditions with their dependencies and have them retrieved and executed automatically upon activation.

6.3.2 Supporting precondition operators

Reuse can possibly be increased further if logical operators can be defined for preconditions. For example, currently the *Extract Method* refactoring has a precondition named *BlockDoesNotUseSuperOrThis*. With a logical *or* operator available, this precondition can be split up into *BlockDoesNotUseSuper* and *BlockDoesNotUseThis*.

6.3.3 Extension to input checking

After checking the preconditions on the user's selection, similar preconditions must be checked on the user's additional input (such as target name for a rename refactoring or target class for a move refactoring). Defining these checks in a similar manner may further promote reuse.

With selection checking however, a single variable is used: the selection inside an active object. User input regularly requires multiple values to be entered by the user. A standard approach would be to store these all in a list or map and pass these to the input precondition checks. Each value will possibly have different preconditions (because they may be method names or class names for example) and a method needs to be developed that maps each value to a subset of the preconditions that are applicable to that value.

6.3.4 Generalizing precondition checking

Libraries and development tools readily exist to extend the Java language with precondition, postcondition and invariant checking, including design by contract libraries (such as iContract [Kramer, 1998]) and aspect-oriented development tools (such as AspectJ [Kiczales et al., 2001]). However, the method proposed by this research only depends on the standard Java language and platform features. It may be worthwhile to extend this method to support more general precondition checking outside the scope of refactoring development, as well as extend it to also support postcondition and invariant checking.

Bibliography

[Bass et al., 2003] Bass, L., Clements, P., and Kazman, R. (2003). *Software Architecture in Practice*. Addison-Wesley, 2 edition.

[Bellon et al., 2007] Bellon, S., Koschke, R., Antoniol, G., Krinke, J., and Merlo, E. (2007). Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591.

[Clements et al., 2003] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J. (2003). *Documenting Software Architectures: Views and Beyond*. Addison-Wesley.

[Deutsch, 1989] Deutsch, L. P. (1989). *Software Reusability - Volume II: Application and Experience*, chapter Design reuse and frameworks in the smalltalk-80 system, pages 57–72. ACM Press.

[Fowler et al., 1999] Fowler, M., Beck, K., Brant, J., Opdyke, W. F., and Roberts, D. B. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

[Gosling et al., 2005] Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005). *The Java Language Specification*. The Java Series. Prentice Hall PTR, 3rd edition.

[IEEE, 2000] IEEE (2000). *IEEE Recommended Practice for Architecture Description*. IEEE Standard 1471.

[Kiczales et al., 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of aspectj. In *ECOOP*, pages 327–353.

[Kramer, 1998] Kramer, R. (1998). icontract - the java design by contract tool. In *Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings*, pages 295–307.

[Lehman, 1996] Lehman, M. M. (1996). Laws of software evolution revisited. In *Proceedings of 5th European Workshop on Software Process Technology*, pages 108–124.

[Mili et al., 1995] Mili, H., Mili, F., and Mili, A. (1995). Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–561.

[Opdyke, 1992] Opdyke, W. F. (1992). *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign.

[Pigoski, 1996] Pigoski, T. M. (1996). *Practical Software Maintenance*. Wiley.

[Roberts, 1999] Roberts, D. B. (1999). *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign.

[Tokuda and Batory, 2001] Tokuda, L. A. and Batory, D. S. (2001). Evolving object-oriented designs with refactorings. *Automated Software Engineering*, 8(1):89–120.

[Widmer, 2008] Widmer, T. (2008). Unleashing the power of refactoring. Website. Last accessed July 8, 2008. `http://www.eclipse.org/articles/article.php?file=Article-Unleashing-the-Power-of-Refactoring/index.html`.

Appendix A

Eclipse JDT Refactoring Architecture

This appendix describes the architecture of the Eclipse JDT Refactoring-component. The description of refactoring development in chapter 2 can be used as an implementation guide to accompany this description.

A.1 Introduction

Software architecture is defined as: "[T]he fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution [IEEE, 2000]." The goal for this description is to provide insight into this fundamental organization. Because there is no industry-standard template for documenting a software architecture, the organization used here is one proposed by Bass et al. since it has proven itself in practice [Bass et al., 2003]. Some changes are however required.

First, the architecture presented here is not constructed but reconstructed: the implementation of the system whose architecture it describes already exists, so instead of prescribing aspects of an implementation, it describes how the system has already been implemented. As a result, the *Architecture background* section is missing, since the original rationale is not available in this reconstruction.

Second, architecture descriptions typically describe an entire system, which is outside the scope of this thesis because this research focusses only on the automated refactoring functionality of the Eclipse JDT. Because of this, the *Context diagram* section is also missing, since the context for every view will be the same: the refactoring functionality is entirely instantiated and operated by and through the Eclipse platform, which itself then forms the entire context.

A.1.1 Component versions

The views are based on Eclipse SDK version 3.3.2 Build id M20080221-1800. The components and their version retrieved from the public Eclipse repository are shown in table A.1.1.

A.1.2 Stakeholder and Viewpoints

The focus of this research is on understanding how refactorings are implemented and how this can be simplified. Therefore, the only relevant stakeholder is the refactoring developer: a software engineer

Component	Version
org.eclipse.jdt.ui	3.3.2_r332_20080128
org.eclipse.ltk.core.refactoring	3.3.2_r332_20080128
org.eclipse.ltk.ui.refactoring	3.3.2_r332_20080128

Table A.1: Component versions used to construct this description.

that needs to implement or maintain a refactoring in the Eclipse JDT environment. The views of the architecture contained in this description are chosen to be useful to this stakeholder.

Typically, three viewpoints are relevant when describing an architecture: (1) the structure as a set of implementation units (module), (2) the structure as a set of elements that have runtime behaviour and interactions (component-and-connector) and (3) the relation to non-software structures in its environment (allocation) [Clements et al., 2003]. The following sections describe the views based on these viewpoints in styles appropriate to the stakeholder.

A.2 Module View

The design of the Eclipse JDT is object-oriented and provides an extension mechanism for refactorings through the use of abstract base classes and interfaces. The class/generalization style is well-suited to express this type of mechanism [Clements et al., 2003] and is therefore used to document this view.

A.2.1 Primary presentation

The primary presentation of the module view is the UML Class Diagram shown in figure A.1. The diagram is standard UML with the exception of the colouring of the objects. For illustration purposes, this diagram includes the relevant classes that make up the *Rename Virtual Method Refactoring*.

A.2.2 Element catalog

Refactoring. Abstract base class for all refactoring implementations. Provides an interface that supports checking preconditions and creating an object representing the required changes. This object can then be applied and reversed (for undo purposes) by the Eclipse environment.

ProcessorBasedRefactoring. Derived from Refactoring. Alternative abstract base class for refactorings that support participants. The actual refactoring functionality is then delegated to a processor. Contains functionality to allow an arbitrary amount of participants to be executed along with the processor and sharing a CheckConditionsContext object.

RefactoringProcessor. Abstract base class for the processor required by a processor-based refactoring. It supports the same interface as the Refactoring base class, although each method signature is extended with a CheckConditionsContext object.

RefactoringParticipant. Abstract base class for a participant. Called by the processor-based refactoring during its execution twice: first after the processor has completed its preconditions checking and again after the processor has created its changes. Both the precondition checking and change creation

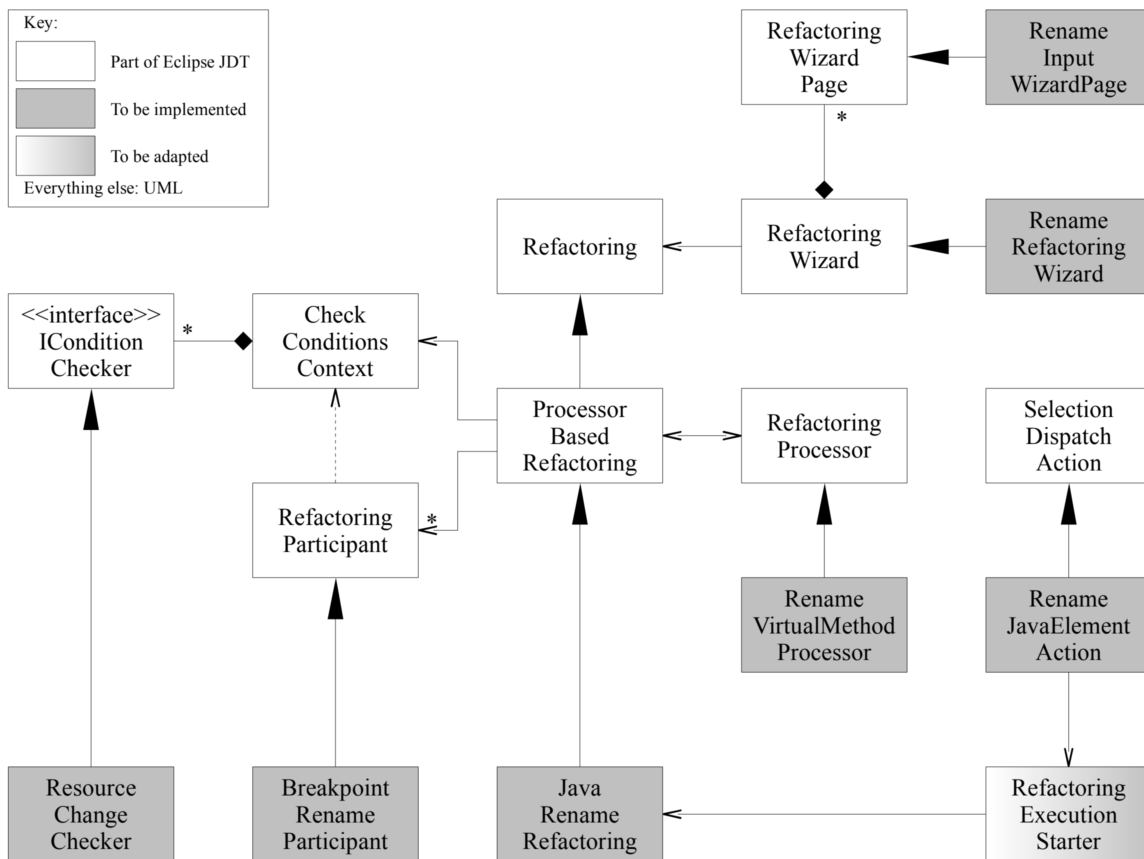


Figure A.1: Parts of the implementation of the Rename Method Refactoring.

can then be extended by each participant by checking additional preconditions or adding additional changes respectively.

CheckConditionsContext. Container class for condition checkers, which are objects that implement the IConditionChecker interface and are shared by a refactoring's processor and participants.

IConditionChecker. Interface for condition checker objects that are shared between a refactoring's processor and its participants in a processor-based refactoring. This sharing allows expensive checks that both the processor and participants require to be shared and executed only once. Examples are checking whether all resources correctly compile or have been saved.

RefactoringWizard. Abstract base class for refactoring wizards. Does not implement any actual user interface functionality, but instead the associated workflow including whether preconditions must be checked and what pages the wizard consists of.

RefactoringWizardPage. Abstract base class for individual refactoring wizard pages. Does not implement any actual user interface functionality, but instead a description of the information and input fields to display on a single page along with functionality to validate the input afterwards.

SelectionDispatchAction. Abstract base class for user interface event handlers involving a selection in the Eclipse workspace. Whenever a user selects an item or changes a selection in the user interface, all registered objects are notified. In case of an action associated with a refactoring, this is typically implemented by performing cheap precondition checks and enabling or disabling the associated

refactoring. When enabled and run, this object launches the refactoring's execution.

RefactoringExecutionStarter. A class commonly used to house code to start a refactoring. A refactoring is typically added by adding a static method that instantiates all related objects (such as wizard, processor, etc.), associates them and then launches the refactoring.

JavaRenameRefactoring, **RenameVirtualMethodProcessor**, **RenameRefactoringWizard**, **BreakpointRenameParticipant**, **RenameInputWizardPage**, **RenameJavaElementAction** and **ResourceChangeChecker** are all classes inheriting from the described base classes or interface implementing (in this case) a refactoring to rename virtual methods and displayed here to illustrate what typically needs to be implemented.

Important to note is that reuse occurs at various levels of granularity: while the RenameVirtualMethodProcessor implements only this refactoring, the JavaRenameRefactoring and RenameJavaElementAction are used by all Java rename refactorings. The ResourceChangeChecker is actually usable by all refactorings.

Furthermore a large amount of classes have been omitted to prevent cluttering the visual representation and comprehensibility of the view. These include mostly base classes such as PlatformObject, which is the root class for nearly all classes for the Eclipse environment.

Additionally, some supporting classes have been omitted that are irrelevant for understanding and using the architecture, such as the RenameSupport class, which instantiates and parameterizes several of the classes in this view (including JavaRenameRefactoring and RenameVirtualMethodProcessor). These classes are not shown because they have no impact on implementing refactorings and understanding the architecture as well as not being required by the architecture.

A.2.3 Variability guide

Eclipse's extension point mechanism, based on an XML configuration file loaded at startup time controls which refactorings are accessible. The binding time for this is *startup time*.

Some parts however are hardcoded and can therefore only be modified by changing the source code and recompiling the component, making their binding time *program write time*. These are:

- **Participant selection**. What set of participants is involved in which refactoring.
- **Condition checker selection**. What set of condition checkers are used by a processor-based refactoring.
- **Several associations**. The coupling between a refactoring, its wizard and pages and its processor (if applicable) are hardcoded in the RefactoringExecutionStarter. The action directly calls the associated method in that class.

A.3 Component-and-Connector View

Execution of a refactoring in the Eclipse environment takes place in the context of a component loaded by the Eclipse platform, providing it with direct access to its services, as well as related services provided by the Eclipse JDT component and other available components. For illustration purposes, this diagram includes the relevant objects that execute when running the *Move Instance Method Refactoring*.

A.3.1 Primary presentation

The primary presentation of the component-and-connector view is the UML Communication Diagram shown in figure A.2. The diagram is standard UML with the exception of the colouring of the objects and the numbering and naming of the messaging. Because the nesting in this diagram is extensive (up to eight levels deep) and the amount of messages high, plain numbering[1] is used and the messages are displayed in table A.3.1. This table includes targets for each message for convenience.

A.3.2 Element catalog

:MoveInstanceMethodAction. The SelectionDispatchAction-derived action class that is executed based on user action: whenever a selection is made in the workspace, the selectionChanged method is called to allow the object to enable or disable its accompanying refactoring. For this refactoring, it is typically enabled when the selection contains a single instance method. When the refactoring is selected by the user, the run method is called. This in turn calls the static startMoveInstanceMethod-Refactoring method in the RefactoringExecutionStarter.

:RefactoringExecutionStarter. A collection of static methods used to tie different parts of a refactoring together by instantiating and associating the relevant classes and then instantiating a RefactoringStarter object and calling its activate method with the creating set of refactoring objects.

:RefactoringStarter. The actual starting point for the execution of a refactoring. Requires instantiated (but not configured) refactoring and wizard objects. Its activate method instantiates a Check-ConditionsOperation object to handle initial precondition checking in a separate thread (the check-InitialConditions method of the Refactoring base class) and if succesful, uses the RefactoringUI to instantiate a GUI dialog to encapsulate the wizard and request the user for input.

:RefactoringUI. A factory for refactoring-related user interface components. Is used here to create a GUI dialog to encapsulate the wizard.

:RefactoringWizardDialog2. One in a set of two possible GUI dialogs to encapsulate the user interface of a refactoring. One is for single-page wizards and the other for multi-page wizards. However, both implementations have evolved to support near-identical behaviour.

:MoveInstanceMethodWizard. The RefactoringWizard-derived wizard class that encapsulates the individual wizard pages that must be presented to the user in order to retrieve input for the refactoring.

:MoveInstanceMethodPage. The RefactoringWizardPage-derived page class that encapsulates the input required from the user to execute the refactoring. Only a single instance is used here: the user must select which class to move the instance method to.

initial:CheckConditionsOperation, **final:CheckConditionsOperation**, **:CreateChangeOperation** and **:UIPerformChangeOperation**. IWorkspaceRunnable-derived operation classes that are used to perform possible long-running operations in a separate thread (Refactoring's checkInitialConditions, checkFinalConditions, createChange and Change's perform methods respectively) to prevent the user interface from becoming unresponsive.

:Change. An objects representing a set of changes to apply to the workspace in order to finalize a refactoring. The change is applied by calling the perform method. However, this method should

[1]UML requires nested calls to be numbered as 1.1.2.1 to make program flow unambiguous.

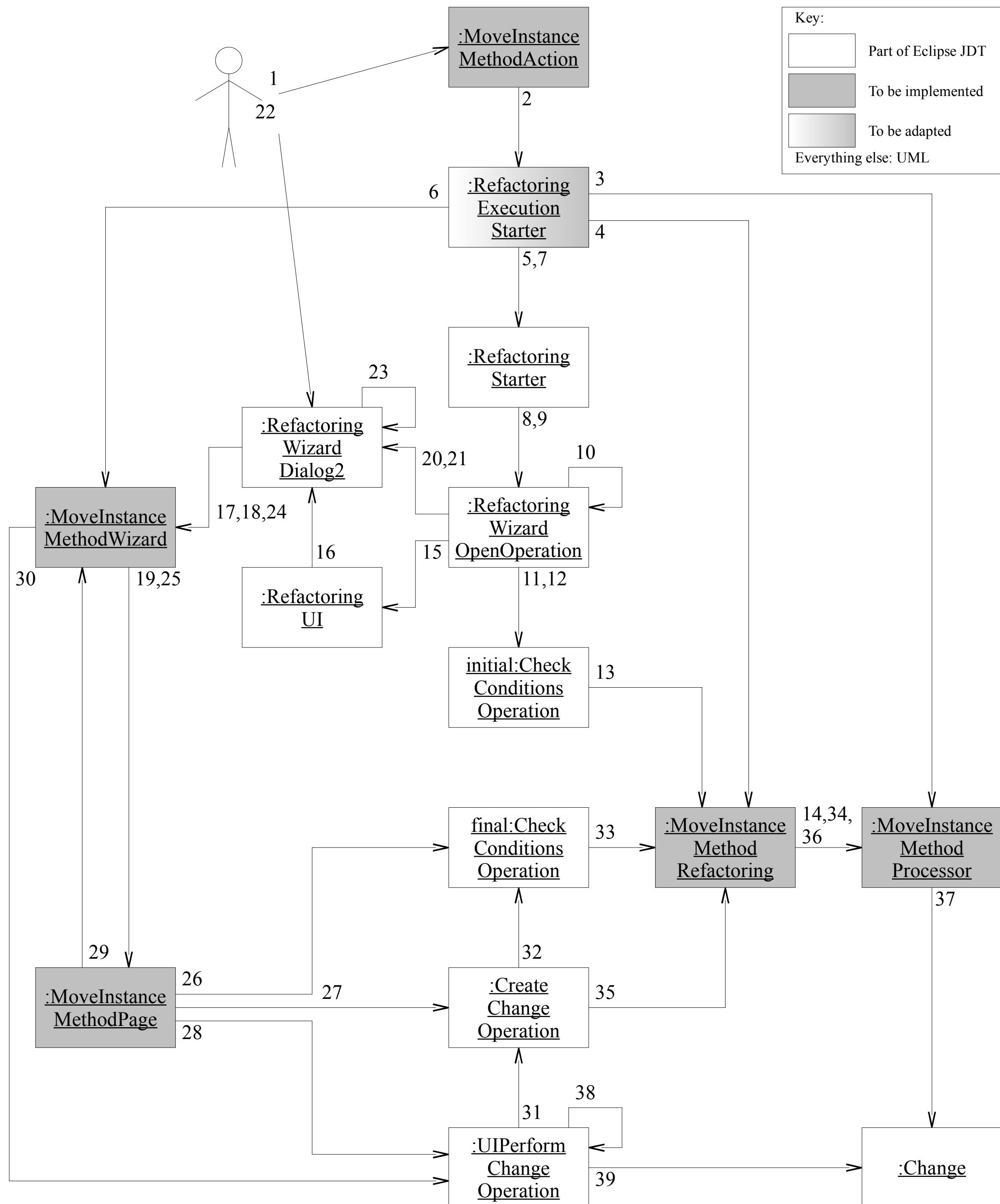


Figure A.2: Executing a refactoring interactively in the Eclipse JDT.

Call	Message	Target
1	run	:MoveInstanceMethodAction
2	startMoveInstanceMethodRefactoring	:RefactoringExecutionStarter
3	<<create>>	:MoveInstanceMethodProcessor
4	<<create>>	:MoveInstanceMethodRefactoring
5	<<create>>	:RefactoringStarter
6	<<create>>	:MoveInstanceMethodWizard
7	activate	:RefactoringStarter
8	<<create>>	:RefactoringWizardOpenOperation
9	run	:RefactoringWizardOpenOperation
10	checkInitialConditions	:RefactoringWizardOpenOperation
11	<<create>>	initial:CheckConditionsOperation
12	run	initial:CheckConditionsOperation
13	checkInitialConditions	:MoveInstanceMethodRefactoring
14	checkInitialConditions	:MoveInstanceMethodProcessor
15	createRefactoringWizard	:RefactoringUI
16	<<create>>	:RefactoringWizardDialog2
17	setContainer	:MoveInstanceMethodWizard
18	addPages	:MoveInstanceMethodWizard
19	<<create>>	:MoveInstanceMethodPage
20	create	:RefactoringWizardDialog2
21	open	:RefactoringWizardDialog2
22	buttonPressed	:RefactoringWizardDialog2
23	okPressed	:RefactoringWizardDialog2
24	performFinish	:MoveInstanceMethodWizard
25	performFinish	:MoveInstanceMethodPage
26	<<create>>	final:CheckConditionsOperation
27	<<create>>	:CreateChangeOperation
28	<<create>>	:UIPerformChangeOperation
29	internalPerformFinish	:MoveInstanceMethodWizard
30	run	:UIPerformChangeOperation
31	run	:CreateChangeOperation
32	run	final:CheckConditionsOperation
33	checkFinalConditions	:MoveInstanceMethodRefactoring
34	checkFinalConditions	:MoveInstanceMethodProcessor
35	createChange	:MoveInstanceMethodRefactoring
36	createChange	:MoveInstanceMethodProcessor
37	<<create>>	:Change
38	executeChange	:UIPerformChangeOperation
39	perform	:Change

Table A.2: Messages and targets for the UML Communication Diagram in figure A.2.

typically only be called by a PerformChangeOperation (or its derived UIPerformChangeOperation) object to prevent the workspace from becoming corrupted.

Base classes are not shown in this diagram, because they have been detailed in the module view already. Also, since the application is GUI-based, additional processing is performed to transform the user input into methods being called on the :MoveInstanceMethodAction and :RefactoringWizardDialog2 objects. This has also been omitted because it has no bearing on the architecture.

A.4 Allocation View

The architecture described has already been implemented. An implementation style allocation view is useful because it describes the mapping from the classes used in the other views to actual configuration items (projects, directories and files) [Clements et al., 2003].

A.4.1 Primary presentation

The primary presentation of the allocation view is the diagram shown in figure A.3. For illustration purposes, this diagram includes the relevant classes that make up the *Extract Method Refactoring*.

A.4.2 Element catalog

Project. A top-level Eclipse plug-in project available from the Eclipse public CVS repository[2]. This translates to a component that is included with the Eclipse JDT when deployed.

Directory. Unrelated to the Java package structure, a sub-directory within the project used to organize code. Projects typically have a **src** directory that contains the package structure, but in the case of **org.eclipse.jdt.ui**, three directories are present, that partition the contents of the complete project into general JDT user interface code, refactoring-related core functionality code and refactoring-related user interface code. Most of the reusable refactoring-related code is located in the **org.eclipse.ltk.*** projects, but as can be seen in diagram A.3, not all.

Package. A standard Java package structure. On disk this is represented as a set of nested directories using the dot as a separator.

File and **File to be implemented**. Actual files on disk. Since the implementation is Java, files are named after the top-level class they contain.

Contains. The relation between the various items. For example, **Refactoring.java** is located in the **org.eclipse.ltk.core.refactoring** namespace, which in turn resides in the **src** directory of the **org.eclipse.ltk.core.refactoring** component.

[2] See http://wiki.eclipse.org/CVS_Howto for information on accessing.

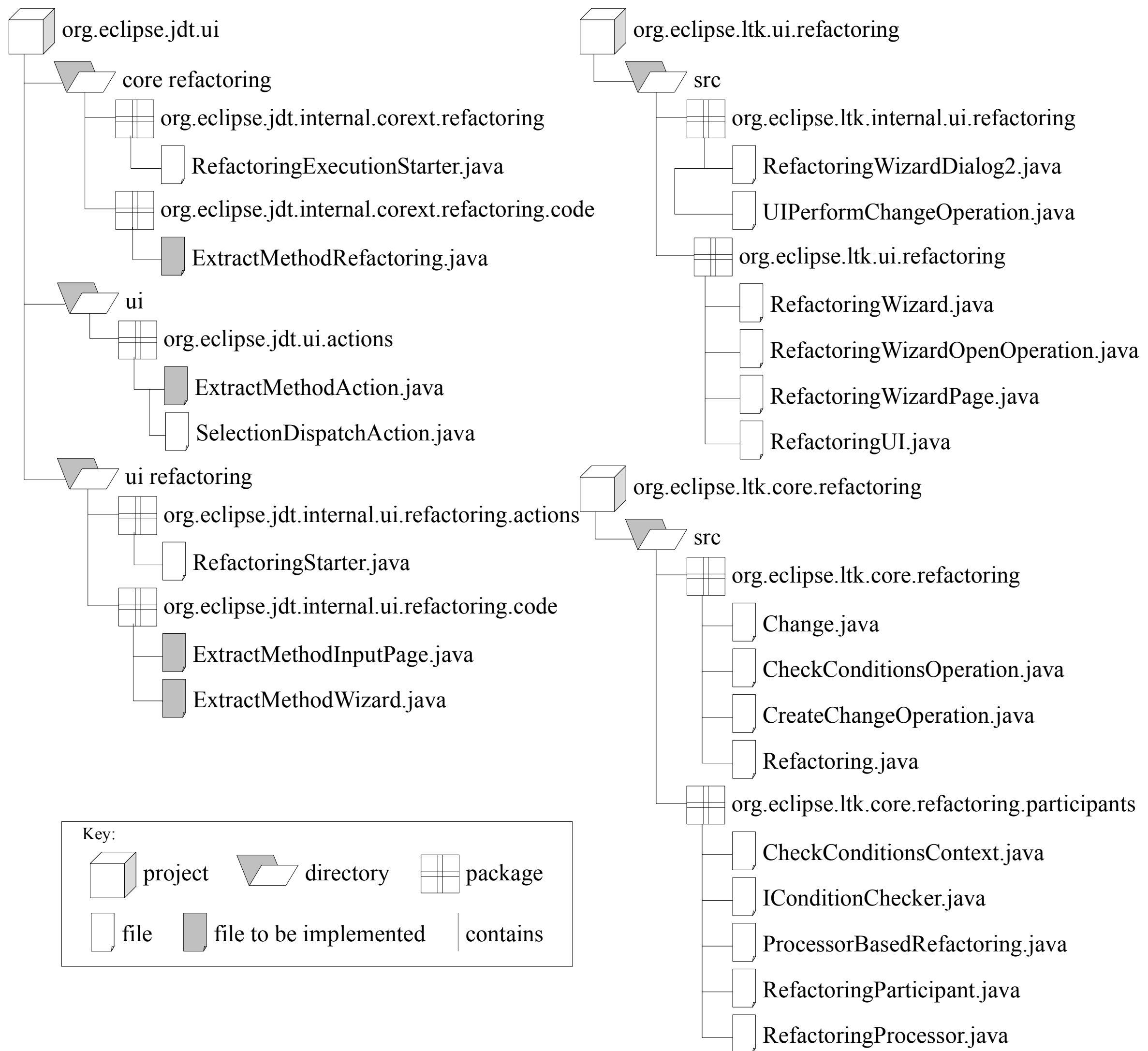


Figure A.3: Project, directory, package and file locations for key classes.

Appendix B

Development Analysis Results

The following tables list the discovered development issues discussed in chapter 3. There is a separate table for each refactoring, listing the type of issue, location and a description of the issue. In the type column, AF refers to Application Flow issue and SC refers to Semantic Clone.

Type (SC/AF)	**Location (class:start-end)**	**Description**
SC	IntroduceIndirectionRefactoring:394-404 RefactoringAvailabilityTester:691:698	Duplicate implementation for checking correct method type.
AF	IntroduceIndirectionAction:115-118	When activated through UI, method type check is performed twice.
SC	IntroduceIndirectionRefactoring:415-416 RefactoringAvailabilityTester:671-672	Duplicate implementation for checking if method is annotation.
SC	IntroduceIndirectionRefactoring:438-439 RefactoringAvailabilityTester:669-670	Duplicate implementation for checking if method is constructor.
AF	IntroduceIndirectionRefactoring:441-449	Setup code unrelated to checking preconditions.

Table B.1: Development issues discovered in the *Introduce Indirection* refactoring.

Type (SC/AF)	**Location (class:start-end)**	**Description**
SC	Checks:728-729 Checks:746-747 RenameMethodProcessor:293	Three implementations to check for compilation errors.
AF	RenameSupport:377	Virtual check is only executed through GUI activation.

Table B.2: Development issues discovered in the *Rename Virtual Method* refactoring.

Type (SC/AF)	Location (class:start-end)	Description
SC	MoveInstanceMethodAction:145-150 RefactoringAvailabilityTester:759-762	Duplicate implementation for retrieving method from text selection.
SC	MoveInstanceMethodAction:93-101 RefactoringAvailabilityTester767-770	Duplicate implementation for retrieving method from structured selection.
SC	MoveInstanceMethodProcessor:1397-1398 MoveInstanceMethodProcessor:2853 RefactoringAvailabilityTester:755	Triple implementation for checking if method is static.
SC	MoveInstanceMethodProcessor:1411-1412 RefactoringAvailabilityTester:755	Duplicate implementation for checking if method is annotation.
SC	MoveInstanceMethodProcessor:1408-1409 MoveInstanceMethodProcessor:2853 RefactoringAvailabilityTester:755	Triple implementation for checking if method is constructor.
AF	MoveInstanceMethodProcessor:1356-1373	Check for recursive call is only executed if unrelated *if* statement is *true*.
SC	MoveInstanceMethodProcessor:2853 RefactoringAvailabilityTester:755	Duplicate implementation for checking if method is binary.
SC	MoveInstanceMethodProcessor:2853 RefactoringAvailabilityTester:755	Duplicate implementation for checking if method is read-only.
AF	RefactoringAvailabilityTester:755	Binary checks are only executed through GUI activation.
AF	RefactoringAvailabilityTester:755	Read-only checks are only executed through GUI activation.

Table B.3: Development issues discovered in the *Move Instance Method* refactoring.

Type (SC/AF)	Location (class:start-end)	Description
SC	ExtractMethodAction:65 ExtractMethodRefactoring:251	Duplicate implementation for zero-length selection check.
AF	ExtractMethodAnalyzer:288-322	Setup code unrelated to checking preconditions.
SC	RefactoringAvailabilityTester:346-363 ExtractMethodAnalyzer:538-598	Duplicate implementation for selection type check.
AF	ExtractMethodRefactoring:270-273	Setup code unrelated to checking preconditions.

Table B.4: Development issues discovered in the *Extract Method* refactoring.

Appendix C

Refactoring Prototype

The following source listings form a working prototype for the proposed changes to the refactoring development for the Eclipse JDT. Most dependencies on the Eclipse libraries have been removed so it can more easily be evaluated as a prototype.

AnnotatedRefactoring.java

```
1 package nl.cwi.sen;
2
3 import org.eclipse.ltk.core.refactoring.RefactoringStatus;
4
5 public abstract class AnnotatedRefactoring {
6     private Object _input;
7     private int _start, _length;
8
9     protected AnnotatedRefactoring(Object input, int start, int length) {
10         _input = input;
11         _start = start;
12         _length = length;
13     }
14
15     protected AnnotatedRefactoring(Object editorInput) {
16         this(editorInput, 0, 0);
17     }
18
19     public final RefactoringStatus checkSelectionConditions() {
20         RefactoringStatus status = new RefactoringStatus();
21         for (ISelectionPrecondition cnd : Precondition.
               getSelectionPreconditions(this)) {
22             RefactoringStatus result = cnd.checkSelectionCondition(_input
                   , _start, _length);
23             status.merge(result);
24         }
25         return status;
26     }
27
28     public final RefactoringStatus checkInitialConditions() {
29         RefactoringStatus status = new RefactoringStatus();
```

```
30         status.merge(checkSelectionConditions());
31         if (!status.hasFatalError()) {
32             for (IInitialPrecondition cnd : Precondition.
                   getInitialPreconditions(this)) {
33                 RefactoringStatus result = cnd.checkInitialCondition(
                       _input, _start, _length);
34                 status.merge(result);
35             }
36         }
37         return status;
38     }
39 
40     public abstract void createChange();
41 }
```

Precondition.java

```
 1 package nl.cwi.sen;
 2 
 3 import java.lang.reflect.Array;
 4 import java.util.ArrayList;
 5 
 6 public enum Precondition {
 7     ContainsNoConstructor(new ContainsNoConstructorPrecondition()),
 8     ContainsInstanceMethod(new ContainsInstanceMethodPrecondition());
 9 
10     private final IPrecondition _checker;
11 
12     Precondition(IPrecondition checker) {
13         _checker = checker;
14     }
15 
16     public IPrecondition getPrecondition() {
17         return _checker;
18     }
19 
20     public static ISelectionPrecondition[] getSelectionPreconditions(
21             AnnotatedRefactoring refactoring) {
22         return getPreconditions(refactoring, ISelectionPrecondition.class
               );
23     }
24 
25     public static IInitialPrecondition[] getInitialPreconditions(
26             AnnotatedRefactoring refactoring) {
27         return getPreconditions(refactoring, IInitialPrecondition.class);
28     }
29 
30     @SuppressWarnings("unchecked")
31     private static <T> T[] getPreconditions(AnnotatedRefactoring
           refactoring,
32             Class<T> type) {
33         Preconditions cnd = refactoring.getClass().getAnnotation(
34                 Preconditions.class);
```

```
35          ArrayList<T> preconditions = new ArrayList<T>();
36          for (Precondition value : cnd.value()) {
37              if (type.isInstance(value.getPrecondition())) {
38                  preconditions.add((T) value.getPrecondition());
39              }
40          }
41          return preconditions.toArray((T[]) Array.newInstance(type, 0));
42      }
43  }
```

Preconditions.java

```
 1  package nl.cwi.sen;
 2
 3  import java.lang.annotation.ElementType;
 4  import java.lang.annotation.Retention;
 5  import java.lang.annotation.RetentionPolicy;
 6  import java.lang.annotation.Target;
 7
 8  @Retention(RetentionPolicy.RUNTIME)
 9  @Target(ElementType.TYPE)
10  public @interface Preconditions {
11      Precondition[] value();
12  }
```

IPrecondition.java

```
1  package nl.cwi.sen;
2
3  public interface IPrecondition {
4  }
```

ISelectionPrecondition.java

```
1  package nl.cwi.sen;
2
3  import org.eclipse.ltk.core.refactoring.RefactoringStatus;
4
5  public interface ISelectionPrecondition extends IPrecondition {
6      RefactoringStatus checkSelectionCondition(Object input, int start,
           int length);
7  }
```

IInitialPrecondition.java

```
1  package nl.cwi.sen;
2
3  import org.eclipse.ltk.core.refactoring.RefactoringStatus;
4
5  public interface IInitialPrecondition extends IPrecondition {
6      RefactoringStatus checkInitialCondition(Object input, int start, int
           length);
7  }
```

ContainsInstanceMethodPrecondition.java

```
 1 package nl.cwi.sen;
 2 
 3 import org.eclipse.ltk.core.refactoring.RefactoringStatus;
 4 
 5 public class ContainsInstanceMethodPrecondition implements
       ISelectionPrecondition,
 6      IInitialPrecondition {
 7 
 8      @Override
 9      public RefactoringStatus checkSelectionCondition(Object input, int
            start, int length) {
10          System.out.println("Checked the selection
                ContainsInstanceMethodPrecondition.");
11          return new RefactoringStatus();
12      }
13 
14      @Override
15      public RefactoringStatus checkInitialCondition(Object input, int
            start, int length) {
16          System.out.println("Checked the initial
                ContainsInstanceMethodPrecondition.");
17          return new RefactoringStatus();
18      }
19 
20 }
```

ContainsNoConstructorPrecondition.java

```
 1 package nl.cwi.sen;
 2 
 3 import org.eclipse.ltk.core.refactoring.RefactoringStatus;
 4 
 5 public class ContainsNoConstructorPrecondition implements
       ISelectionPrecondition {
 6 
 7      @Override
 8      public RefactoringStatus checkSelectionCondition(Object input, int
            start, int length) {
 9          System.out.println("Checked the selection
                ContainsNoConstructorPrecondition.");
10          return new RefactoringStatus();
11      }
12 
13 }
```

MoveInstanceMethodRefactoring.java

```
 1 package nl.cwi.sen;
 2 
 3 import static nl.cwi.sen.Precondition.*;
 4 
 5 @Preconditions({ ContainsNoConstructor, ContainsInstanceMethod })
```

```java
6 public class MoveInstanceMethodRefactoring extends AnnotatedRefactoring {
7
8     public MoveInstanceMethodRefactoring(Object input, int start, int
         length) {
9         super(input, start, length);
10     }
11
12     @Override
13     public void createChange() {
14         System.out.println("Executing MoveInstanceMethodRefactoring...");
15     }
16
17 }
```

Program.java

```java
1 package nl.cwi.sen;
2
3 import org.eclipse.ltk.core.refactoring.RefactoringStatus;
4
5 public class Program {
6
7     public static void main(String[] args) {
8         AnnotatedRefactoring mim = new MoveInstanceMethodRefactoring(null
             , 0, 0);
9         RefactoringStatus status = new RefactoringStatus();
10         status.merge(mim.checkInitialConditions());
11         if (!status.hasFatalError()) {
12             mim.createChange();
13         }
14     }
15
16 }
```

Using these source files, precondition checks can be implemented by implementing either the *ISelectionPrecondition* or *IInitialPrecondition* interface (or both) and adding them to the *Precondition* enumeration as the included examples. Refactorings can be implemented by extending the *AnnotatedRefactoring* class and by adding the *Preconditions* annotation referencing all applicable preconditions.