A Refactoring Code Browser For Java

Final Report

Graham Bennett

June 2002

Supervisor: Susan Eisenbach
Second Marker: Jeff Magee

Individual Project
MEng Computing Degree

Department of Computing
Imperial College
Abstract

This paper presents a framework for refactoring Java code. A refactoring is an operation which modifies code to improve its design without affecting the meaning of the program. The framework presented is designed to be integrated into a particular development environment.

The framework is designed to encourage a particular structure in the refactorings that are implemented in it. In particular, it encourages explicit modelling of pre and post conditions, and of individual changes made. It is proposed that this will improve correctness and code reuse. The framework also caters for composite refactorings, that is refactorings which are composed of other refactorings.

A tool has been produced to illustrate how the framework can be used. Work has also been done on integrating the framework into existing tools such as the Emacs environment.

Several primitive refactorings have been implemented, as well as some composite ones, to demonstrate that the framework is functional and useful. Work has also been done on the concept of refactorings which introduce design patterns to code. Refactorings have been implemented which do this.
Acknowledgements

I would like to thank my supervisor Susan Eisenbach for her invaluable help and guidance throughout the project, and for proposing such an interesting project in the first place. Thanks must also go to my second marker Jeff Magee for his helpful advice, and to the staff and students who attended our regular group meetings.

I would also like to thank my family and friends for their support, especially my brother Philip for his professional proof reading.

Finally I would also like to thank the developers of the various libraries and tools that I have used - most notably those of the Transmogrify library - and the refactoring community in general.
Contents

Abstract i

Acknowledgements iii

Contents v

1 Introduction 1
  1.1 Motivation ......................................................... 1
  1.2 Contribution .................................................... 2
  1.3 Report Structure .................................................. 2

2 Background 3
  2.1 The State of the Art ............................................. 3
  2.2 Research ......................................................... 3
    2.2.1 Existing Work ............................................ 3
    2.2.2 Online Resources .......................................... 12
    2.2.3 Existing Tools ............................................ 12
    2.2.4 Tools and Libraries ...................................... 15
  2.3 Prototypes ....................................................... 16
  2.4 Evaluation ....................................................... 16
    2.4.1 Framework Implementation ................................ 16
    2.4.2 User Interface ............................................. 18
    2.4.3 Analysis and Helper Functions ........................... 18
    2.4.4 Modelling Change ........................................ 19
    2.4.5 Conclusion ................................................ 19
  2.5 Summary ......................................................... 20
3 Design

3.1 A Definition ..................................... 21
3.2 Goal ........................................... 21
3.3 Refactoring Java Programs ...................... 22
   3.3.1 Language Features .......................... 22
   3.3.2 Difficulties ................................ 23
   3.3.3 The Refactoring Process .................... 24
3.4 Requirements .................................... 24
3.5 Implementation Decisions ........................ 25
3.6 Overall Architecture ............................ 26
3.7 Components .................................... 27
   3.7.1 Refactorings ................................ 27
   3.7.2 Conditions .................................. 27
   3.7.3 Code Model ................................. 31
   3.7.4 Changes .................................... 31
   3.7.5 Tool Interface ............................... 31
3.8 User Interface ................................... 32
3.9 Summary ......................................... 33

4 Implementation ................................. 35

4.1 Framework ..................................... 35
   4.1.1 Code Model ................................. 35
   4.1.2 Refactorings ................................ 37
   4.1.3 Conditions .................................. 37
   4.1.4 Changes .................................... 39
   4.1.5 Patterns .................................... 39
   4.1.6 Errors and Exceptions ....................... 40
4.2 The Tool Interface .............................. 41
   4.2.1 Components ................................ 41
   4.2.2 The Refactoring Process .................... 44
   4.2.3 Handling User Input ......................... 45
   4.2.4 Handling Changes and Undo ................. 45
   4.2.5 File Handling ............................... 47
   4.2.6 Managing Printing ......................... 47
4.3 Prototype GUI Tool ............................. 47
### Features
- 4.3.1 Features
- 4.3.2 Interface
- 4.3.3 Architecture
- 4.3.4 Problems Encountered

### Emacs Tool
- 4.4 Emacs Tool
  - 4.4.1 Interfacing Between Java and Emacs
  - 4.4.2 Architecture
  - 4.4.3 Problems Encountered

### XML Refactoring Description
- 4.5 XML Refactoring Description
  - 4.5.1 Architecture
  - 4.5.2 The XML Description
  - 4.5.3 Problems Encountered

### Libraries
- 4.6 Libraries

### Summary
- 4.7 Summary

### Testing
- 5 Testing
  - 5.1 Testing Strategy
  - 5.2 Unit Testing
    - 5.2.1 Examples
  - 5.3 Functional Testing
    - 5.3.1 Strategy
    - 5.3.2 An Example Test Case
  - 5.4 A Refactoring Example
    - 5.4.1 Pull Up Field
    - 5.4.2 Pull Up Method
    - 5.4.3 Encapsulate Field
    - 5.4.4 Extract Method
    - 5.4.5 InlineTemp
    - 5.4.6 IntroduceSingleton
  - 5.5 Summary
## 6 Conclusions

6.1 Evaluation ............................................. 83

6.1.1 Analysis ............................................. 84

6.2 Contributions .......................................... 86

6.3 Further Work .......................................... 87

6.3.1 Automated Application of Refactorings ............ 87

6.3.2 User-Driven Refactoring Composition ............... 88

6.3.3 XML Description of Refactorings .................... 88

6.3.4 Extensions for J2EE .......................... 89

6.4 Summary ............................................. 89

### List of Figures

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>91</td>
</tr>
</tbody>
</table>

### References

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>95</td>
</tr>
</tbody>
</table>

### A Availability of Sources and Documentation

A.1 Sources ............................................. 99

A.2 Documentation ......................................... 99

A.3 Binaries ............................................. 100

### B Refactorings

B.1 Notation ............................................. 101

B.2 Rename Method ........................................ 102

B.3 Inline Temp ........................................... 102

B.4 Pull Up Field .......................................... 103

B.5 Pull Up Method ......................................... 104

B.6 Change Package ....................................... 104

B.7 Extract Method ......................................... 105

B.8 Encapsulate Field ...................................... 106

B.8.1 Add Accessors ....................................... 106

B.8.2 Replace Field Accesses With Queries .............. 107

B.8.3 Protect Field ........................................ 107

B.9 Introduce Singleton ..................................... 108

B.9.1 Add Singleton Instance ................................ 108

B.9.2 Replace Instantiations With Singleton Accesses .... 109

B.9.3 Enforce Single Instance ............................. 109
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.10</td>
<td>Introduce Memento</td>
<td>110</td>
</tr>
<tr>
<td>B.10.1</td>
<td>Create Memento Class</td>
<td>111</td>
</tr>
<tr>
<td>B.10.2</td>
<td>Add Create Memento Operation</td>
<td>111</td>
</tr>
<tr>
<td>B.10.3</td>
<td>Add Set Memento Operation</td>
<td>112</td>
</tr>
<tr>
<td>C</td>
<td>User Guide</td>
<td>115</td>
</tr>
<tr>
<td>C.1</td>
<td>Prototype GUI Tool</td>
<td>115</td>
</tr>
<tr>
<td>C.1.1</td>
<td>Requirements</td>
<td>115</td>
</tr>
<tr>
<td>C.1.2</td>
<td>Installing and Running the Tool</td>
<td>115</td>
</tr>
<tr>
<td>C.1.3</td>
<td>Configuring the tool</td>
<td>115</td>
</tr>
<tr>
<td>C.1.4</td>
<td>Setting up a project</td>
<td>116</td>
</tr>
<tr>
<td>C.1.5</td>
<td>Opening a project</td>
<td>117</td>
</tr>
<tr>
<td>C.1.6</td>
<td>Compiling a project</td>
<td>117</td>
</tr>
<tr>
<td>C.1.7</td>
<td>Applying a refactoring</td>
<td>118</td>
</tr>
<tr>
<td>C.1.8</td>
<td>Testing a refactoring</td>
<td>118</td>
</tr>
<tr>
<td>C.1.9</td>
<td>Undoing a refactoring</td>
<td>119</td>
</tr>
<tr>
<td>C.2</td>
<td>Emacs Tool</td>
<td>119</td>
</tr>
<tr>
<td>C.2.1</td>
<td>Requirements</td>
<td>119</td>
</tr>
<tr>
<td>C.2.2</td>
<td>Installation</td>
<td>119</td>
</tr>
<tr>
<td>C.2.3</td>
<td>Configuration</td>
<td>120</td>
</tr>
<tr>
<td>C.2.4</td>
<td>Applying a Refactoring</td>
<td>120</td>
</tr>
<tr>
<td>C.2.5</td>
<td>Testing a Refactoring</td>
<td>120</td>
</tr>
<tr>
<td>D</td>
<td>Developer Guide</td>
<td>121</td>
</tr>
<tr>
<td>D.1</td>
<td>Documentation</td>
<td>121</td>
</tr>
<tr>
<td>D.2</td>
<td>Tool Developers Guide</td>
<td>121</td>
</tr>
<tr>
<td>D.2.1</td>
<td>Implementation of the Tool Interface</td>
<td>121</td>
</tr>
<tr>
<td>D.2.2</td>
<td>Use of the other tool classes</td>
<td>122</td>
</tr>
<tr>
<td>D.2.3</td>
<td>The Refactoring Process</td>
<td>123</td>
</tr>
<tr>
<td>D.3</td>
<td>Framework Developers Guide</td>
<td>124</td>
</tr>
<tr>
<td>D.3.1</td>
<td>Primitive Refactorings</td>
<td>124</td>
</tr>
<tr>
<td>D.3.2</td>
<td>Composite Refactorings</td>
<td>124</td>
</tr>
<tr>
<td>D.3.3</td>
<td>Conditions</td>
<td>125</td>
</tr>
<tr>
<td>D.3.4</td>
<td>Changes</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>XML Refactoring Descriptions</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>-----------------------------</td>
<td>---</td>
</tr>
<tr>
<td>E.1</td>
<td>RenameMethod</td>
<td>127</td>
</tr>
<tr>
<td>E.2</td>
<td>Document Type Definition</td>
<td>128</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This project is about refactoring Java programs in an automated way through the creation and use of tools.

1.1 Motivation

Refactoring was conceived to deal with one of the major problems in software development: change. As software ages, the requirements the users have for it change. This means the software must change to meet these new needs. As the software is changed more and more, it becomes increasingly difficult to add new features until a point is reached where the software has to be redesigned from scratch.

Refactoring is a disciplined way of restructuring code in order to improve the design while preserving the meaning. With a refactoring approach, one does not simply add a new feature to the software. Instead, one reorganises - that is refactors - the code so that the feature can be added easily, and then adds the feature.

However there are several problems with the refactoring process. One is that, until recently, it has been an essentially manual process. The other is the issue of safety - a refactoring should preserve the meaning of the code. Although each refactoring is applied in a disciplined way, it is easy to make mistakes. There is therefore a need for a tool to perform refactorings automatically and safely.

There have been several attempts in recent years to create automated refactoring tools. This has been most successful in the Smalltalk community, where the Refactoring Browser has become an essential tool for the Smalltalk developer.

There have been several efforts to create a similar tool for the Java language. These however have not met with as much success. There are several reasons for this. Firstly, in order to be used, the refactoring tool has to be well integrated with the development environment. In the Java community there are a plethora of development environments in use, unlike in the Smalltalk com-
munity. Some of these do offer refactoring capabilities, but they are not as well developed as they could be.

It is unreasonable to suppose that a refactoring tool could be produced which the whole Java community would begin to use as their development environment. There is therefore a definite need for a refactoring tool that is independent of - but not separate from - the development environment. This tool could then be integrated into development environments as required, and work could still be centralised on a single refactoring framework.

1.2 Contribution
This project contributes a novel framework for creating refactorings in a structured and formal way, and for applying them to source code in a safe and automated fashion. Included in this framework is a library of useful refactorings, many of which are unique to this project. The framework has been designed to be extensible, and to be independent of both platform and development environment. It provides high level support for composite refactorings and refactorings which introduce design patterns, and supports innovative features such as the ability to specify refactorings in XML.

The other main contribution is a tool which builds on the framework to create a usable refactoring code browser and editor. It enables the user to apply refactorings safely to Java programs, and its undo and safety testing features mean that it can be used to perform speculative design. The framework has also been ported to the Emacs development environment.

1.3 Report Structure
Chapter 2 reviews existing work in the refactoring field. This provides motivation for the design stage of the project, which is described in Chapter 3. Chapter 4 describes how the various components of the project were implemented and Chapter 5 describes how the implementation was tested.

Of particular note is section 5.4, which gives an example of how the system presented might be used. It shows a simple Java program and how it can be improved using refactorings that have been implemented.

Chapter 6 analyses the system that has been presented and evaluates the worth of the work that has been done in this project. It also analyses what further work might be done to build on the work presented here.

A complete list of the refactorings which have been implemented can be found in Appendix B. Appendices C and D give documentation for users of the prototype tool and developers who wish to work on the framework or integrate the framework into other tools.
Chapter 2

Background

2.1 The State of the Art

In the last few years several refactoring tools have become available for the Java language. These are usually integrated into a popular IDE (Integrated Development Environment). They are able to perform low-level refactorings such as the renaming of identifiers and the moving of methods.

The idea of composite refactorings was conceived several years ago, but has not been widely implemented. There are some tools which are the result of research, but none are widely used by developers, with the exception of the Smalltalk Refactoring Browser[1].

The concept of refactoring to Design Patterns[2] has also been researched as an extension to composite refactorings. Design Patterns describe elements of a systems design which follow a well-known pattern. It is proposed that recognising the applicability of these patterns and using them where appropriate will improve code quality and make the functionality of the code more recognisable.

Few refactoring tools give the capability to introduce Design Patterns into code, and those that do do not take a composition approach, rather they tend to use an ‘all or nothing’ method and do not differentiate much between low and high level refactorings.

There is an evident need for a Java refactoring tool that is independent of development environment but that can nevertheless be integrated with one. There is also a need for a refactoring tool that implements more of the ideas that have been researched recently, to do for Java what the Refactoring Browser did for Smalltalk.

2.2 Research

2.2.1 Existing Work

Much work has been done on the topic of refactoring since the early work done by Opdyke[3] and others. This has taken the form of books, academic papers
Refactoring - Martin Fowler

Martin Fowler’s refactoring book[4] serves several useful purposes. It gives an excellent introduction to the topic of refactoring - the principles behind it and its uses. It also contains an extensive catalogue of refactorings, each with a method for application and an example. It also contains many guest contributions from well known people who work in the field, including Kent Beck, John Brant, William Opdyke and Don Roberts.

The book makes a good argument for the use of refactoring as a programming paradigm, especially in the context of Extreme Programming[5]. In this methodology, design work is performed after implementation has commenced, and so refactoring is essential as a way of making design changes on the existing code.

The book also highlights the importance of testing in refactoring. It advocates the use of JUnit[6] for unit testing. It emphasises the importance of the tests being ‘self-checking’, i.e. they return yes or no results indicating whether the test succeeded or failed and require no work on the part of the tester to check. It is envisaged that this testing strategy will be used in the development of this project.

Another idea mentioned in the book is the need to refactor in small steps, in order to detect errors as quickly as possible and avoid propagating them. This is motivation for the idea of composite refactorings, which will be will be investigated in detail in this project.

Reading this book highlighted several possible extensions to the project, including:

- **Code Smell Analysis**
  This is an idea proposed by Kent Beck which tries to formalise the idea of ‘bad smells’ in code. That is structures and conventions which indicate badly written or buggy code to another programmer reading it.

  A useful possible extension to the project would be to create a tool which could use the techniques described to detect ‘bad code’ and use this information to highlight for the programmer areas where problems might arise and refactorings could be applied.

- **Duplicate Code Elimination**
  It is easy to see that duplicated logic in an object-oriented environment is a bad thing. If the logic for a particular operation is duplicated in several places, a change which needs to be made to the algorithm used will need to be made in several places. It is therefore easy for the programmer to miss instances of the operation and introduce bugs.

  It would be useful to construct a tool which could detect these duplications and highlight them to the programmer. The programmer could then choose to apply refactorings in order to remove the duplication. There have been several attempts to do this in practice[7, 8, 9].
2. Background

Refactoring Object-Oriented Frameworks[3]

This paper is considered the seminal work in the refactoring field, and was the first analysis that resulted in a practical refactoring tool. Opdyke’s tool was a refactoring tool for C++.

One contribution of this work was to identify several different approaches to tool-based refactorings. One approach described is the one used by many developers before the advent of automated tools. It involves applying the refactoring and recompiling the code. If an error occurs the refactoring is rolled back. This approach has several disadvantages. Firstly the recompilation process maybe very long. In the context of composite refactorings, which this project will focus on, many small refactorings may need to be made to achieve a high level refactoring. This would therefore involve many recompilations and so a possibly large amount of time. Another flaw, which is probably the most damaging, is that the compiler is not able to catch all the problems which a refactoring might introduce. Successful compilation only guarantees syntactic correctness, not preservation of meaning.

Another approach is one that has been adopted by most automated tools. It involves computing a set of preconditions for the refactoring which, if satisfied, will guarantee that the refactoring will preserve the meaning. This approach removes the dependence on the compiler to check correctness. The paper also advocates giving informal arguments for meaning preservation for the user to refer to.

Opdyke talks about composite refactorings in his paper in the context of high-level refactorings which can be constructed from smaller ones. However he treats them in exactly the same way as low-level refactorings. They would have their own set of preconditions and would be used much like a single large monolithic refactoring.

Opdyke’s task was complicated by the fact that he had chosen C++ as his target language. C++ has explicit pointers which can be modified at the language level. This causes many problems when reasoning about correctness of a refactoring. C++ also allows multiple inheritance which could complicate refactorings which change the inheritance model of a program. In order to combat some of these problems, Opdyke introduced some constraints for the programs his system could refactor reliably:

- Unique superclass
- Distinct class names
- Distinct member names
- Inherited member variables not redefined.
- Compatible signatures in redefinition of member functions.
- Type safe assignments
- Semantically equivalent references and operations.

The choice of Java as the target language eliminates some of these problems, such as multiple inheritance and explicit pointers. Some of the other constraints can be satisfied by specifying that a program to be refactored must
compile correctly before refactoring. The final constraint is the most important. This was Opdyke’s attempt at formalising the requirement of meaning preservation.

Opdyke introduced a way of formalising the preconditions that must be imposed before a refactoring can be applied. He created functions which could be used in predicate expressions to formalise constraints. These are analogous to the Analysis Functions used later by O’Cinneide[10] and Roberts[11].

**Practical Analysis for Refactoring[11]**

Don Roberts' thesis describes a practical method of describing the refactoring process and its implementation in the Smalltalk Refactoring Browser[1]. It builds on Opdyke's work, placing more focus on automatic checking of behaviour preservation and composition of refactorings.

One major contribution is the new definition of refactoring which is presented. He considers that any significant refactoring (i.e. one which goes beyond renaming of identifiers) cannot by definition be strictly behaviour preserving, as it will change the execution of the program in some way. In some environments, for example a real-time application, this will change the behaviour of the program. Also some refactorings are designed to specifically change a program’s behaviour in order to improve it.

Roberts therefore presents a new weaker definition of a refactoring: a program transformation that has particular preconditions that must be satisfied before the transformation can be applied, and a set of postconditions that will hold afterwards.

The notion of postconditions is a major step, as it eliminates much of the complex analysis that needs to be done when refactorings are composed and performed in sequence. This is because, in a composite refactoring, the preconditions for primitive refactorings are often ‘set up’ by the preceding refactorings.

Another area that Roberts covers is the computation of dependencies between refactorings. This has several possible applications:

- Ability to undo refactorings which are part of a sequence
- Conflict detection
- Parallelisation of independent refactorings

Roberts presents a method of deciding whether one refactoring depends on another. It is based on the concept of commutativity between two refactorings in a chain. If the chain \( \langle R_1, R_2 \rangle \) does not commute then \( R_2 \) depends on \( R_1 \).

An algorithm for separating chains of independent refactorings from a larger chain is also presented.

Roberts also covers the idea of dynamic analysis of refactorings. This involves using the actual execution of the program to determine whether behaviour has been preserved. If not, the refactoring is undone. In this situation the test suite for the program is critical, as the tests must exercise all possibilities for changes in behaviour to be detected at run-time. This approach can solve the problems associated with static analysis such as high complexity or over-conservative

---

*Research*  
2. **Background**
2. Background

static approximations, but is problematic in itself since it relies on good tests and the ability to execute the program.

Roberts also notes the analogy between introduction of Design Patterns to a program and refactoring. He demonstrates this with a refactoring to introduce the Strategy pattern to a program.

The paper then moves on to discuss practical issues with the implementation of a refactoring tool. Initial prototypes of the Smalltalk Refactory revealed the following practical requirements:

- **Speed**
  The refactoring process must be faster than refactoring manually, and fast enough to allow exploratory use of the tool.

- **Undo**
  This feature is essential if the user is to use the tool in an exploratory fashion.

- **Integration into a development environment**
  Developers will find the tool less useful if it is separate from their everyday development environment.

Finally the paper discusses the design of the Smalltalk Refactoring Browser. Briefly, the system has six main components:

- Refactorings - the refactorings themselves
- Conditions - pre and post conditions
- Changes - the changes made by a refactoring
- Smalltalk Parser - the low level language parser
- Tree Rewriter - used to integrate changes into the syntax tree
- Formatter - used to generate formatted code from the tree

The design of the Smalltalk Refactoring Browser is quite generic and has many parallels with tools for other languages such as Java.

**Composite Refactorings for Java Programs[10]**

This paper serves as a convenient summary of the composite refactorings aspect of O’Cinneide’s later PhD thesis[12].

The paper provides motivation for the idea of Composite Refactorings, that is high-level refactorings made up of smaller, simpler low-level refactorings. O’Cinneide proposes that this approach gives you several advantages, the main one being that it is far easier to argue behaviour preservation for the individual small refactorings than for the entire high-level refactoring.

In this model, each refactoring (whether primitive or composite) has a precondition and a postcondition. If the precondition holds, the refactoring may be applied safely. If this is done correctly the postcondition will hold and the program’s behaviour will be preserved. The postcondition serves as a formal description of the effect of the refactoring. O’Cinneide uses a notation similar to
that of Don Roberts[11] to express these conditions. An informal ‘preservation argument’ is also included as a weaker ‘proof’ of behaviour preservation.

One of the main contributions of this paper is the presentation of a technique to calculate the pre and post conditions of an overall refactoring from the conditions of the primitive refactorings which make it up.

The system presented uses no internal representation of the program itself. It simply uses so-called Analysis Functions to derive information from the program. Helper Functions are also used to derive more high level information. These functions have their own pre and post conditions, but have no side effects on the program.

The main difficulty involved in such a system is therefore the computation of the pre and post conditions for an overall refactoring. O’Cinneide identifies two methods for composing refactorings: chaining and set iteration. Chaining refactorings corresponds to composing them in sequence. The postcondition of one refactoring will help to set up the precondition of the next. Set iteration corresponds to applying a refactoring to several components ‘in parallel’. The refactoring is applied to a given set of program elements, for example ‘all methods’ or the result of some kind of selection expression.

Chaining
O’Cinneide recognises that it is not sufficient to simply conjoin the preconditions of all the primitive refactorings to obtain an overall precondition. One reason for this is that part of a precondition for a refactoring some way down the chain may be set up by a preceding refactoring, and so will not necessarily be true until that refactoring has been applied.

The computation for a chain therefore consists of two steps:

- **Legality test and precondition computation**
  The precondition is, put simply, calculated by evaluating, for each sub-chain \((R_1, R_2)\) of the chain, the parts of the precondition of \(R_2\) which are not guaranteed by the post condition of \(R_1\) and ANDing them with the precondition of \(R_1\). If a contradiction arises at any point then the chain is illegal.

- **Postcondition computation**
  Postconditions are calculated by what O’Cinneide calls ‘sequential AND-ing’. This involves, for each sub-chain \((R_1, R_2)\), ANDing the two postconditions together. If a contradiction arises, the part of the postcondition of \(R_1\) which causes the contradiction is dropped.

Set Iteration
Here refactorings can take place in any order. O’Cinneide recognised that a sufficient legality test is to check if any precondition depends on a postcondition of another refactoring, i.e. the precondition does not hold unless the postcondition holds. If this is the case there is a definite ordering for the two refactorings, and so set iteration cannot be legal as this order might not be maintained.

The computation for set iteration therefore consists of three steps:
2. Background

- **Legality Test**
  This involves checking there are no dependencies as explained above.

- **Precondition Computation**
  It is sufficient to simply AND all the preconditions together. This is because the precondition of the first part of the chain must form part of the overall precondition, and in set iteration any refactoring can be the first part of the chain.

- **Postcondition Computation**
  A similar argument can be applied for the postcondition of the overall chain.

O’Cinneide’s work builds on that of Don Roberts, but improves on it in the following ways which are useful in this project:

- Explicit calculation of legality.
- Computation of pre and post conditions of overall refactorings. This allows the refactoring to be used in later composite refactorings.

The framework allows the pre and post conditions of an overall refactoring to be computed statically and reused.

**Automated Application of Design Patterns - A Refactoring Approach[12]**

This paper includes the work explained in O’Cinneide’s previous paper, and then goes on to explain a methodology for developing refactorings which introduce design patterns. That is, a way of using transforming a program to conform to a particular design pattern in an automated fashion using refactoring techniques.

O’Cinneide defines four criteria which are motivation for this methodology:

- Behaviour preservation
- Applicability to real programs
- Reuse where possible
- Programmer controls quality

In order to accomplish his task, O’Cinneide introduces several concepts. First is the concept of a Precursor. This can be loosely defined as ‘where the design pattern transformation begins’. It can be thought of as pattern in the design of the existing program which is both necessary and sufficient to warrant the introduction of the pattern. The factory pattern is used to illustrate this. A precursor for the introduction of the factory pattern is a relationship between two classes: a creator and a product. The factory method pattern is introduced in order to support creation of different product classes.

Once a precursor has been identified, the next step is to break the pattern down into minipatterns. These are primitive patterns, analogous to a primitive refactoring. The idea is that all design patterns are composed of these minipatterns. O’Cinneide suggests that only a small number of these minipatterns are necessary to express most patterns. In fact his thesis defines only six.
For each minipattern there is an associated minitransformation. A minitransformation is constructed by hand using the minipattern and the precursor as a guide. It is analogous to a composite refactoring, in that it is constructed from primitive refactorings and has pre and post conditions.

O’Cinneide’s methodology for automation of refactoring to a design pattern is therefore the following:

- Select the pattern to be used
- Identify a precursor
- Decompose the pattern into minipatterns
- Define the minipatterns in terms of minitransformations, reusing minitransformations as much as possible
- Define the high-level transformation as a composition of minitransformations

In this way O’Cinneide defines a practical way of performing design pattern transformations using a refactoring approach.

**A Meta-Model for Language-Independent Refactoring[13]**

The main motivation for this paper is the commonality between refactoring approaches for various languages. It proposes a meta-model for the representation and application of refactorings in a language independent way.

Programs are modelled at the so-called Program Entity Level, that is:

- There is no exact control flow analysis
- It is sufficiently abstract to allow for different languages.

The main components of the proposed framework are:

- A language independent analysis component.
- A language specific interpretation module, one for each language.
- A language specific analysis module, one for each language.
- A common front end.

This paper is useful from the point of view of design - it gives some inspiration on how a generic refactoring framework might be designed. However a language independent solution is probably beyond the scope of this project.

**Elbereth: Tool Support for Refactoring Java Programs[14]**

This paper introduces a refactoring tool based on the notion of ‘star diagrams’. A star diagram provides a structured view of a program’s source code as a network of nodes. It hides information not relevant to the data structure and maintenance of the code, and groups similar elements into ‘node stacks’. This allows the developer to easily see where possible redundancy is present in the program.
The result of this paper is a refactoring tool for the Java language - Elbereth - which has been derived from an earlier implementation for procedural programs. The tool implements some simple refactorings, but could be more useful as a tool for finding possible points in the program where a refactoring could be applied.

This paper provides some useful information on what kind of refactorings might be implemented and how refactorings might be presented to the user. It does, however, concentrate on how potential refactorings might be identified in a program, which is not a part of the core features of the project.

**Tool Support for Object-Oriented Patterns[15]**

This paper discusses the issues around developing a tool which introduces design patterns into software developed in the Smalltalk language. It makes several useful observations which are useful in the design of a refactoring browser, regardless of whether it has any design pattern capability.

One useful observation is the choice of interface to the tool. In this case the developer is given three different views of the software - pattern, design and code views. The pattern view allows the user to choose a pattern to apply from a repository. The design view presents the user with a graphical representation of the class hierarchy which can be edited. The code view is simply the source code of the particular class being worked on.

In order to represent patterns, the tool uses the notion of ‘fragments’. A fragment is a generic notion of a program element, be it a class, a method, a pattern occurrence or a relation between two fragments. The pattern repository is simply a collection of prototype fragments which can be cloned and applied to the existing code.

One other interesting aspect is exception handling. When an error occurs while applying a transformation, the tool will raise an exception. An interesting consideration is how to handle the error. The authors implemented several different handlers, for example one that attempts to repair the defect, one that gives the user a choice of what the next action should be, one that simply warns the user and continues, and one that simply ignores the problem.

This paper provides a lot of material on user interface design for a refactoring tool - information which will be useful in the design of the prototype tool.

**Design and Development of a Program Transformation Tool[16]**

The main focus of this paper is to develop a catalogue of program transformations - both low and high level - which are practically useful. The paper then goes on to describe the implementation of a refactoring tool for Java - the DPT (Design Pattern Transformer).

There is not a large amount of information on the implementation of this tool. It is however based on the idea of using primitive low-level refactorings to transform a program to conform to a particular design pattern. This is a similar strategy to that proposed for this project.

The tool which was created as part of this paper is reviewed later in Section 2.2.3.
2.2.2 Online Resources

The Refactoring Homepage[17]

The Refactoring Homepage is a website created by Martin Fowler to accompany his refactoring book. It also serves as a useful reference to the various refactoring resources on the web. The catalogue of refactorings from the book is also available here, and has been added to since the book was published. A useful resource for this project is the list of available refactoring tools.

The homepage also links to an article written by Martin Fowler called 'Crossing Refactoring's Rubicon'[18]. This discusses the fact that, during 2001, several tools became what Fowler describes as ‘serious’ refactoring tools. These tools are distinguished by the fact that they implement some of the less straightforward refactorings such as ‘Extract Method’.

Industrial Logic: Refactoring to Patterns[19]

This is a catalogue of refactorings in a similar vein to Fowler’s[17] which involve refactoring to design patterns. The information given is of course not written with automated application of the refactorings in mind, but it is nevertheless an excellent resource for the design of a tool which intends to perform refactoring to design patterns.

The Refactoring Mailing List[20]

The refactoring mailing list has over six hundred subscribers and hosts regular discussions of many different aspects of refactoring. It was very useful during the research stage of this project, especially in determining the current ‘state of the art’ at the outset.

The Refactoring Tools Wiki[21]

This wiki is a useful repository for discussion of the various refactoring tools available. There is also a Java Refactoring Tool wiki[22].

2.2.3 Existing Tools

Several refactoring tools have emerged recently. As part of the research for this project, some of the most prominent have been evaluated. The results are given below.

Smalltalk Refactoring Browser[1]

The Smalltalk refactoring browser was the first refactoring tool to be widely used. It implements most of the standard class, method and field refactorings for the Smalltalk language.

Smalltalk is a very clean language which lends itself more to automatic processing than other languages such as, say, C++. This is probably one of the reasons this tool has been so successful. Java however is also quite clean and unambiguous, and this is the main reason for the recent interest in Java refactoring tools. Java is used much more in industry than Smalltalk, so it is possible
2. Background

a Java refactoring tool might become more popular than its original Smalltalk counterpart.

**Eclipse[23]**

The Eclipse project aims to create a ‘universal tool platform’. It is backed by many industry leaders. At the core is a language-independent development environment which offers some refactoring features:

- Rename local variable, parameter
- Move method
- Extract method
- Self-encapsulate field

Eclipse is interesting because it aims to be a general development environment, rather than focusing on Java. However Java is the only language it supports properly at the moment. It is probably for this reason that the refactoring support is currently quite limited. Having said that it does enough to cross the rubicon[18].

**Idea[24]**

Idea is a commercial development environment created by IntelliJ. 

Idea has the following refactoring features:

- Rename package, class, method, field, method parameter and local variable
- Move class and package
- Extract Method
- Introduce Variable
- Change Method Signature

It also has a wealth of other features and tools which could not be described as performing refactorings, such as a code formatter and an advanced search engine.

Idea does not implement many refactorings, and could not be described as a refactoring browser. However, those refactorings it does have are very well implemented, and the emphasis is placed on everyday usefulness.

**JFactor[25]**

JFactor is a plug-in for IBM Visual Age for Java. It is one of the most complete refactoring tools for Java currently available.

Its refactoring features include:

- **Method Refactorings**
  Extract, rename, pull up, push down, inline, introduce foreign.
- **Field Refactorings**
  Rename, pull up, push down, encapsulate.
• **Class Refactorings**
  Extract superclass, extract interface.

JFactor’s main strength is its good selection of refactorings. Like many dedicated refactoring tools it is built as a plug-in for an existing development environment, in this case Visual Age.

**Design Pattern Transformer[26]**

This tool was the result of Alexej Kupin’s masters’ thesis[16]. Its main focus is the introduction of design patterns via refactoring, although it does also implement some of the lower level refactorings such as renaming and moving fields and methods.

The main difference between this and other tools is the interface. Instead of being code-oriented as most tools are, it is very much oriented to the particular refactoring or pattern that the user wants to apply or introduce. Once the operation has been chosen, the user is only then asked what program elements to apply the operation to. This is in contrast with other tools where the user will select the program element first.

This approach makes sense in the context of this tool as its main focus is the introduction of design patterns. When a user wishes to introduce a design pattern to existing code, the required program elements may not currently exist, or at least may not be recognisable when compared to their equivalents after refactoring.

The tool uses the concept of ‘catalogues’ of refactorings and patterns which can be extended by others. The user selects a particular catalogue and then a particular operation to apply from a tree structured menu. The user creates a project which holds all the source files to be refactored.

Unfortunately this tool is quite under-developed and does not seem to have been worked on since the thesis was originally produced.

**Transmogrify[27]**

Unlike most of the other tools, Transmogrify is not simply a refactoring tool. The project aims to create a framework for parsing and modifying Java source code. To date it has several components:

• Symbol table and AST (Analytic Component)
• Refactoring Framework
• Hook Interface for tools
• A Lint Tool and ‘Code Twiddler’

The refactoring framework has several refactorings implemented for it, including:

• Rename Symbol (variable or method)
• Extract Method
• Replace Temp With Query
The refactoring aspect of Transmogrify is not particularly well developed, however the underlying framework is more interesting. The Analytic framework is well separated from the rest of the project and is designed to be used as a library in other projects.

Transmogrify is still under development. In particular it does not handle a small subset of the more complicated Java expressions. However it does include features to detect whether it is able to understand a particular block of code.

2.2.4 Tools and Libraries

There are many tools and libraries available which might be useful in this project. If libraries are available to perform a particular task it is usually more useful to try to build on these libraries rather than re-implement from scratch.

One specific area in which use of an external library is important is that of parsing the Java code itself. The libraries available fall into two categories:

- **Parser Generators** - these can be used to create a parser which will parse Java code and create an AST (Abstract Syntax Tree) based on the code. The information store in this structure is in general just simple syntactic information. Of these libraries ANTLR and SableCC have been examined.

- **Analytic Frameworks** - these are based on some kind of parser which they use to generate AST. They then analyse the AST data to extract information on (for example) types, definitions and references. Of these libraries JParse and Transmogrify have been examined.

**ANTLR[28]**

ANTLR (ANother Tool for Language Recognition) is a parser generator which has grammars for several languages including Java. It is one of the more well-known and well-developed parser systems, especially since it has grammars for many different languages.

**SableCC[29]**

SableCC is another parser generator library for Java. It is quite widely used in the Java community, and also has grammars for several different languages.

**JParse[30]**

JParse builds on the ANTLR parser to produce a parser, type evaluator and exception analyser for Java. JParse therefore adds type and exception information to the basic AST produced via ANTLR. This information would be useful in the construction of the framework to be presented in this project.

However JParse does not seem to be extensively developed and is also not synchronised with the most recent versions of ANTLR.
Prototypes

2. Background

Transmogrify Analytic Framework[27]

Transmogrify is similar to JParse in that it builds on ANTLR. It adds information about the type, definitions and references to identifiers in the AST. It therefore gives the developer a lot more useful information than a plain parser or a library such as JParse.

Transmogrify also seems to have been developed to a relatively high standard and has been proven to be usable since it is used in the Transmogrify refactoring system as well as other tools.

2.3 Prototypes

Evaluation of Transmogrify Refactoring Framework

The purpose of this prototype was to make use of the Transmogrify refactoring framework to carry out simple refactorings. In order to achieve this a simple Java program was written which used the hook interface to play the role of a tool in order to request a particular refactoring.

The following conclusions were drawn:

- The tool hook interface is too monolithic and is not quite general enough to abstract the tool away from the framework.
- The model of a refactoring used is not general enough. That is there is no distinction between the conditions and the changes caused. This makes it difficult to compose refactorings and to undo changes made.

Evaluation of Transmogrify Analytic Framework

A simple program was created that would parse a given file using Transmogrify and then print the source back out from the AST. This was used to test several aspects:

- The Transmogrify parsing facilities
- The Transmogrify printing facilities
- The usability of the Transmogrify API in general

The tests were successful on the whole, as Transmogrify was able to produce code equivalent to the original. However there were differences in layout and formatting, as would be expected.

2.4 Evaluation

2.4.1 Framework Implementation

The background research for the project has highlighted three main approaches to implementation. These are:
2. Background Evaluation

Integrate with an existing IDE

This is the approach that many of the non-commercial refactoring tools have taken to date. There are several advantages to this approach:

- Little or no work is needed for the GUI.
- There is a pre-existing API for manipulating the syntax tree and possibly also code generation.

However there are also some disadvantages:

- The existing API may not be sufficient and modification might be problematic, especially if the tool is not open-source.
- The tool would be less portable, less separable from the IDE.
- Commercial IDEs are often heavyweight applications, which might make development work difficult.

Create a new system from scratch

The advantages of this approach are:

- Full control over every aspect of the tool and framework.
- The framework can be made as general and as portable as desired.

The disadvantages are:

- The GUI needs to be created from scratch.
- There is no parsing or printing framework to build on.

Create a new tool based on existing libraries

This approach would involve using a pre-existing library such as Transmogrify or JParse. The advantages of this approach are:

- Near-complete control over the design
- Independence of IDE
- The framework can still be integrated with an IDE if necessary
- The parsing and printing framework (if they exist) are completely open and can be extended if necessary.

The disadvantages are:

- The GUI may still need to be created from scratch.
- The libraries used may not be complete and might need work to extend them.
2.4.2 User Interface

Since the aim of this project is to build a refactoring code browser for Java, it will be necessary to create a code browsing tool which uses the framework which will be made.

Evaluation of the tools currently available has indicated several different approaches to user interface design:

- **Code-oriented interface**
  This is the approach taken by most IDEs. The user is presented with a view of the project’s source code, and can choose to apply refactorings by selecting program entities from within the code and choosing the refactoring to apply.

- **Design-oriented interface**
  This approach involves presenting the user with a design view of the project. This usually takes the form of a class hierarchy interface. The user chooses program elements to refactor from within the hierarchy.

- **Refactoring-oriented interface**
  In this approach, the user first chooses which operation they wish to perform. They are then presented with the necessary information to perform the operation.

- **Combined interface**
  This approach combines all three of the previous approaches, giving the user the option of which view on the code they would like. This approach is advocated by Florijn et al. in their paper[15].

2.4.3 Analysis and Helper Functions

Analysis functions and helper functions are a common theme in refactoring implementations. They are used both to reason logically about refactorings and in the implementation.

**Analysis Functions**

Analysis functions have two roles - they are used as mathematical functions in logical expressions, and also have an implementation counterpart.

Sometimes the result of the analysis is undecidable. O'Cinneide suggests three ways of solving this:

- **No implementation actually necessary**
  The condition has already been established by a previous refactoring. In this case the system might be aware of this from a database of known facts and so evaluation may continue.

- **An estimation can be made**
  A conservative estimation can be made, which may make some feasible refactorings impossible but can ensure safety. Roberts describes a dynamic analysis methodology to help solve this[11].
2. Background

- User input
  The system may seek the input of the user in order to solve the problem. As O’Cinneide states[12], the programmer would have to consider the precondition if refactoring manually, so their workload is not increased.

Roberts defines primitive (e.g. isClass) and derived (e.g. subclasses) analysis functions. He also defines AST functions.

This project will combine these approaches. Analysis functions will be used in logical reasoning, and will have implementation equivalents in the form of Condition objects.

Helper Functions

These operations are used to extract more complicated information, for example a set of methods to be used in an interface. These are analogous to Roberts’ derived analysis functions. O’Cinneide provides his helper functions with pre and post conditions to aid reasoning.

Helper functions in this project will be implemented in the underlying framework, as part of the particular component to which they refer.

2.4.4 Modelling Change

The research stage has shown that it is better to model changes to the code rather than to make them directly. This approach has several advantages:

- Changes can be repeated, stored and modified.
- It aids implementation of undo/redo.
- It simplifies the separation between the refactoring framework and the AST implementation.

2.4.5 Conclusion

The tool produced in this project should be complex enough to cross the refactoring rubicon[18].

Of the three implementation options given above, the third option seems the most attractive. This involves basing the tool on existing libraries. After evaluating existing software, it seems that the Transmogrify framework is most suitable for this purpose.

A stand-alone tool will be created for the project. It will use the framework’s tool interface to perform the refactorings. The interface employed will be of a combined nature as described previously.

The main reason for choosing to implement a tool from scratch is so that interface limitations between the tool and the framework do not constrain the design of the underlying system as it develops. Once the framework is fully developed it can be integrated with other tools.
2.5 Summary

After reviewing the work that has been done already, one can conclude that it is within the scope of this project to create a framework which is close to the current state of the art. There are several tools and libraries which might be employed to avoid re-implementing work which has already been done and allow the project to progress more quickly.

Research has highlighted several points where decisions must be made about how a particular part of the framework should be implemented. The merits of the various options have been examined and decisions will be made in the design section to follow.
Chapter 3

Design

3.1 A Definition
Before any design work can be attempted, a practical definition of refactoring should be agreed upon. Traditionally, refactoring has been defined as follows:

Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure. [4]

Roberts however takes a different view[11]. In his definition, a refactoring simply consists of a set of preconditions which must be satisfied before the transformation is applied, and a set of post conditions which will hold afterwards if this is the case. This definition admits refactorings which change the behaviour of the program, but do so in a well-defined way.

The refactorings presented in this report will be defined in terms of their pre and post conditions. In most cases, these conditions will be an argument for the behaviour preservation of the transformation. However we will not restrict ourselves to this - we will allow refactorings which intentionally modify the behaviour of the program in a well-defined and useful way.

In all cases the central concept is that the pre conditions must hold in order for a refactoring to be applied, and if so the transformation will in all cases ensure that post conditions hold after application.

3.2 Goal
The goal of this project is to create a refactoring tool for the Java language. That is to implement a refactoring framework, and to build a usable refactoring tool based on that framework.

The design is intended to encourage a more structured approach to the refactoring process. In detail:
- Explicit distinction and reuse of conditions
- Separation and reuse of distinct changes
- Directly undoable changes

It is thought that this more structured approach will have several beneficial effects:

- Encourage code reuse between refactorings
- Encourage simplicity in the implementation of refactorings
- Improve the chance of correctness

3.3 Refactoring Java Programs

There are many practical considerations which must be taken into account when designing a refactoring tool for the Java language.

3.3.1 Language Features

The Java language has many advantages from the point of view of automated processing. It is strongly typed and has a well defined syntax and program layout. This means that ambiguous program code is avoided. Also there are in general no specialised ‘flavours’ of Java, and code conforms to a single specification[31].

These features make source modification and generation much simpler. Importantly, in Java one can always identify a program element using some fully qualified name. This is a great advantage when processing program elements in an automated way. This is the main reason why it must be stipulated that the code to be refactored must compile correctly. Successful compilation gives a good indication that the code is well specified Java, and so it can safely assumed that all the properties given above hold.

In terms of the actual structure of the language, there are several advantages in the Java approach but also several potential difficulties. In Java there is no multiple inheritance, a property which simplifies refactorings which change the inheritance model of the program. However interfaces are a special case. While multiple implementation inheritance is not allowed, Java permits multiple interface inheritance. Interfaces can therefore have multiple supertypes, and classes can implement multiple interfaces. Refactorings therefore have to take special care when dealing with interfaces. In some cases the refactoring is meaningless when applied to an interface, in others stronger preconditions must be asserted in order for safety to be preserved.

Another feature that needs to be examined is the use of modifiers. Modifiers are used to provide extra information about program elements such as classes, methods and class members. Some modifiers are used to specify the visibility of the element to other elements (public, private, etc.). In this case refactorings must ensure that their operation does not reveal information that was
previously restricted, and that sensible decisions are made for the visibility of new elements which are created.

Other modifiers such as the abstract keyword constrain the structure of the program. Refactorings must take care to obey these constraints in order to produce valid programs. Modifiers such as synchronized and transient have more subtle implications as discussed in the next section.

Another important feature is the notion of packages. Packages allow the developer to group classes into a common namespace. Apart from being an organisational aid, there are also implications in terms of visibility. Elements can be defined as 'package-private', meaning they are only accessible to other elements in the package. Refactorings therefore need to be aware of packages. When introducing, moving or removing program elements it must be ensured that no visibility problems are created and that the correct packages are imported to ensure that the code is still correct.

Exceptions are another important consideration. Methods in Java can be defined to 'throw' exceptions. If such a method is overridden in a subtype then certain requirements are placed on the method's throws clause. Refactoring implementations must take care to obey these rules when manipulating methods.

### 3.3.2 Difficulties

Although Java is very suited to automated processing, there are still some difficult aspects.

One is the use of modifiers such as synchronized. For example, adding or moving a synchronised method might be done in a way which preserves meaning in the context of a single execution thread. However in the presence of multiple threads this transformation might alter the behaviour significantly by causing deadlock or race conditions. It is not feasible for a tool such as the one presented here to test for this type of situation, in fact this type of test might even prove intractable. In these cases we fall back to our definition of refactorings in terms of pre and post conditions and leave the programmer to make an estimation of the applicability of the refactoring.

Another is the use of the Reflection API. This allows a Java program to discover information about loaded classes and operate on them at runtime. For example one can instantiate an object of any type depending entirely on the input to the program, and call methods in a way which cannot simply be analysed statically from the program code. Again, full analysis of this might well be intractable, and so use of reflection will not be considered in this project.

Extensions defined in the Java 2 Enterprise Edition[32] introduce new constraints on program code. This would cause problems for a refactoring tool which tried to preserve the meaning of these programs. For example there are restrictions placed on a class which defines an Enterprise JavaBean: file handling is forbidden, as is passing this as a parameter and the use of threads. A refactoring tool would have to respect these restrictions. Additionally, in an enterprise context the program code does not completely define the semantics
of the program - other information is provided through XML-based deployment descriptors. A refactoring tool would therefore also have to parse and understand these descriptors. This project will not consider these problems, and will concentrate on the Java 2 Standard Edition. While this does not necessarily prevent it being used in an enterprise context, the onus will be on the developer to ensure that the refactorings applied do not affect the correctness of the program with respect to the J2EE specification. This is however an area for possible further work, as discussed later.

3.3.3 The Refactoring Process

In simple terms, the process of refactoring Java code will be modelled as follows. A representation of the code will first be obtained by parsing the relevant source files. Once this has been done, the representation will be analysed to check that the preconditions of the refactoring hold. If this is the case, the representation will be modified in order to implement the refactoring. Once this has been done, the post conditions should now hold if they were to be tested. The final task is then to convert the representation back into source form. The resulting source code should be valid Java code which compiles correctly.

3.4 Requirements

The following are the basic requirements of the overall framework:

- **Primitive Refactorings**
  The software should have support for primitive refactorings such as Rename Method, Pull Up Field.

- **Composite Refactorings**
  The software should incorporate a framework for composing primitive refactorings to create high level refactorings.

- **Undo Ability**
  The user should be able to undo refactorings safely to allow exploratory use of the system.

- **Usability**
  The software should provide a usable graphical user interface. It should also meet the fundamental usability requirement that using the tool must be faster and more effective than refactoring manually, although this may be difficult to measure.

- **Simple Addition of New Refactorings**
  The software should provide a framework which supports addition of new refactorings (both primitive and composite).

- **Tool Independence**
  The framework should be independent of the environment in which it is integrated. This will enable it to be integrated in multiple different development environments.
3.5 Implementation Decisions

There are several decisions which must be made before design of the system can begin. These have been discussed in general terms in the background chapter.

Whether to build on existing tools or start from scratch

The conclusion that has been drawn is that it is better to separate the framework completely from the tool which uses it. A tool can then be created from scratch which allows full access to the framework’s features whilst being completely separable from it.

However, as Don Roberts states in his thesis[11],

"The most important criteria are the ones which integrate the refactoring process with other tools."

The framework should therefore provided facilities to integrate easily with other development tools. It might also be feasible to demonstrate this with a particular development tool if time allows.

The project will therefore aim to create a stand-alone tool which is a true ‘Refactoring Code Browser’ for the Java language, to serve as a prototype. The framework can then be integrated with other tools at a later time.

How to represent refactorings within the framework

The approach that will be taken is to separate the refactoring into component parts, drawing on the designs given by Roberts[11] and O’Cinneide[10]. The design should separate at least these aspects of a refactoring:

- The conditions (both pre and post)
- The action of the refactoring, i.e. the changes it makes.

How to parse the Java code itself

The Transmogrify[27] analytic framework will be used to parse the Java code. This extends the ANTLR[28] parser with a more convenient representation. There are some aspects of the framework which are not currently fully implemented. Some work may need to be done on improvement in these areas. Changes can then be submitted back to the Transmogrify team.

This approach is preferable to using the AST and parsing facilities of an existing tool, because a lot of indirection would be necessary to abstract the framework away from being dependent on the tool’s specific facilities. With this solution the Transmogrify framework will be used regardless of what tool the system is integrated into.
How code generation should be handled
The Transmogrify analytic component also contains facilities for printing and formatting code from the AST. Preliminary tests indicate this is functional but not complete, and so some extension of this aspect might be necessary.
Again this is preferable to using tool specific facilities to accomplish code generation, for the same reasons given above.

How should the interface with tools be modelled?
The interface will be mainly based on the Façade design pattern. The hook interface approach used by Transmogrify is considered too monolithic and not abstract enough for the purposes of this project. Communication between the tool and the framework will be modelled using the Observer pattern.

How should the User Interface be modelled?
The research phase of the project has indicated that a combined interface would be the most useful.

How should Undo functionality be implemented?
Since each refactoring has both pre and post conditions in the proposed model, it should be possible to allow the user to undo refactorings in an arbitrary order. Provided the conditions are strong enough this should not pose a threat to the safety of the tool. This would require changes to be undone in a systematic fashion, by applying the necessary changes to reverse the refactoring.
However one problem that was identified early on is that some refactorings are very difficult to undo in such a systematic fashion. For example, the In-lineTemp refactoring replaces all references to a temporary variable with the value it was initialised to. It is not a simple task to record these changes in a way that allows the code to be subsequently modified and the change still to be undone. It will therefore be necessary to use a different approach to undo this type of refactoring.

3.6 Overall Architecture
Figure 3.1 shows an outline of the whole system. The framework will have five main components:
- The refactorings themselves
- The code model
- Conditions
- Changes
- The tool interface

Section 3.7 gives details of each of these components.
The other major component will be the tool itself, described in section 3.8.

3.7 Components

This section gives a more detailed specification for the various components of the system. The UML diagram (Figure 3.2) shows how the various components fit together.

3.7.1 Refactorings

This component represents the refactoring itself. It is envisaged that it will be based on the Gang of Four Composite pattern[2], with a subclass for each primitive refactoring and a subclass representing composite refactorings. See Figure 3.3. Application of a refactoring will result in a Change object, which can then be applied to the code model.

3.7.2 Conditions

This component is also based on the Composite pattern. Primitive conditions will be composed by the logical conjunction condition (the AND composite) - see Figure 3.4. The condition object will have a method to determine whether the condition it represents holds or not. In order to determine this it will communicate with the code model. This communication will be modelled on the Visitor design pattern.

There will be a subclass of the Condition object representing each condition that needs to be expressed. Each one of these will be the analogue of a mathematical analysis function.

An important aspect of the condition component is the algorithm used to calculate the condition of a composite refactoring. The algorithm used to compute the pre and post conditions will be an adaptation of that of O’Cinneide and Roberts. The Strategy pattern will be employed to allow the algorithm to vary independently of the rest of the system.

Figure 3.1: Framework Overview
3. Design

Components

transmogrify

AST

SymTab

Printer

Refactoring

preConditionAlgorithm : ConditionAlgorithm
postConditionAlgorithm : ConditionAlgorithm

+apply() : Change
+getPreCondition() : Condition
+getPostCondition() : Condition

Condition

+holds(model:CodeModel) : boolean

ConditionAlgorithm

+calculate(in PrimitiveRefactoring:) : Condition
+calculate(in CompositeRefactoring:) : Condition

CodeModel

+parse() +check(condition:Condition) +change(change:Change) +print()

ConditionAlgorithm

+calculate(in PrimitiveRefactoring:) : Condition
+calculate(in CompositeRefactoring:) : Condition

AST

Condition

SymbolTable

Printer

Figure 3.2: Overview
3. Design Components

Refactoring
preConditionAlgorithm : ConditionAlgorithm
postConditionAlgorithm : ConditionAlgorithm
+apply() : Change
+getPreCondition() : Condition
+getPostCondition() : Condition

PrimitiveRefactoring
+apply() : Change
+getPreCondition() : Condition
+getPostCondition() : Condition

CompositeRefactoring
+apply() : Change
+getPreCondition() : Condition
+getPostCondition() : Condition

Figure 3.3: Refactoring

Figure 3.4: Model of Conditions
Figure 3.5: Condition
3.7.3 Code Model

This component will encapsulate the abstract syntax tree, symbol table, code formatter and related components. It will interface directly with the Transmogrify framework.

It is not envisaged that it will be necessary to separate the AST and Symbol Table implementation from this component, as Transmogrify will be an integral part of the entire framework. If this becomes necessary the component can be refactored. The structure of the Code Model component can be seen in 3.6.

![Figure 3.6: Code Model](image)

The Code Model will have a method which takes a Change object and applies the changes therein. This interaction will be based on the Visitor pattern.

The code model will also contain facilities to format and print the contents of the AST after the change has been applied. This will use the Transmogrify Printer component, although this will probably have to be subclassed and improved.

3.7.4 Changes

Performing a refactoring results in Change objects. These can then be applied to the code model (see Figure 3.7). There will be a different class of Change for each primitive change that can be made by a refactoring. The Composite pattern will be employed to allow high level changes to be represented as a composition of primitive changes.

The Change objects will be modelled on the Command pattern. This gives several advantages, including the ability to store and repeat changes and simplifying implementation of an undo feature.

3.7.5 Tool Interface

The tool interface forms the layer between the framework and the tool itself - see Figure 3.8. The communication through this interface will be two way, since the tool will need to make requests to the framework and the framework will
need to notify the tool. The Façade and Observer patterns will be employed to support this communication.

The tool interface will need to support the following interactions:

- Requests for user input and the associated response.
- When the user initiates a refactoring, a method for determining the currently selected program element which the refactoring should be performed on.
- Facilities for displaying messages and errors to the user.
- An interface for determining which source files are currently in use.

The tool interface will therefore export an API (Application Program Interface). Code can then be written based on this API to interface a given tool with the framework.

### 3.8 User Interface

The user interface will be a stand-alone tool which uses the tool interface component to interface with the framework. It is envisaged that the tool will be relatively simple but complex enough to allow all of the refactoring features to be used to their full extent.

The tool component will also serve as a method of testing the framework during development, so it is envisaged it will evolve in parallel with the framework.
The research phase has shown that a good refactoring tool needs to have an interface that is both refactoring-oriented, design-oriented and code-oriented. The tool will therefore exhibit what we have termed a combined interface - it will present multiple different views of the software to the user depending on the operation being performed.

It is envisaged these views will consist of:

- A design view. This will present the user with a class hierarchy view of the software.
- A code view. This will present an editable view of the source code.
- A refactoring view. This will be used to gain the user’s input when a refactoring is being performed.

See Figure 3.9 for an outline of these views.

3.9 Summary

In summary, a framework will be created which will have a generic interface to tools which use it. Parsing and code generation will be handled by the framework, not the tool. The framework will try to enforce a more formal structure on refactorings. The project also aims to create a stand-alone tool which demonstrates the capabilities of the framework in a usable refactoring tool.
Figure 3.9: User Interface
Chapter 4

Implementation

This chapter gives details of the various issues that were encountered during the development of the project, and the various approaches that were taken. UML diagrams are given to describe the main classes involved, although they do not show complete information for the sake of clarity.

The project has been given the name ‘Freefactor’, which was chosen as the conjunction of free and refactor. It is intended that the project will be released under the GNU General Public License[33] and be available freely when it is completed.

4.1 Framework

The refactoring framework has been constructed as a Java library which builds upon the open source Transmogrify library. Transmogrify in turn builds upon the ANTLR parsing system to provide facilities for parsing and generating Java code.

The main component of the framework is the CodeModel, which encapsulates the various components needed parse the code, as well as the representation of the code once it has been parsed. It therefore provides an interface to the parser itself, as well as components such as the AST and the symbol table. It also encapsulates various components to analyse, create and modify the code in the AST, namely the CodeAnalyser, the CodeMaker and the CodeManipulator.

4.1.1 Code Model

The code model represents code within the framework. The model provides simple methods to parse and print the code, and stores information about the code that has been parsed, including the AST and symbol table. See Figure 4.1 for details of the CodeModel and related classes.
Code Analyser

This component is used mainly by Conditions to check various properties of the code. Properties to be checked are represented by individual methods. An instance of the code analyser exists for each code model, and will examine the contents of the code model without modifying it.

For example, operations exist in the CodeAnalyser to find the local variables in a method and get the signature of a method. These functions mostly operate on the AST stored in the CodeModel.

Code Manipulator

The Code Manipulator contains methods to manipulate fragments of the AST, for example adding and removing children from a particular node. It is intended that all parts of the framework should use this for all AST manipulation in order to avoid errors.

Manipulating tree components is a very common task throughout the system. Each AST node has various pointers for the node’s parent, children, next sibling and previous sibling. If these are manipulated manually it is very easy, for example, to forget to set one of the parent or child pointers to the correct value for a particular node. The Code Manipulator ensures this is done correctly.

Code Maker

This component is used to create new blocks of code. It has no association with the code model itself, as the new constructs created are independent of the existing code. It does however use an instance of the Code Manipulator in order to manipulate new AST components.

Figure 4.1: Code Model and Related Classes
4. Implementation Framework

4.1.2 Refactorings

Refactorings are represented using the Composite design pattern, which allows them to be composed together to form high level refactorings. Each refactoring has a precondition and a postcondition, which is specified in terms of Condition objects. The pre and post conditions for composite refactorings are computed using a particular ConditionAlgorithm. Figure 4.2 shows the representation of a refactoring in detail.

The following refactorings have been implemented thus far:

- PullUpField
- PullUpMethod
- EncapsulateField
  - AddAccessors
  - ProtectField
  - ReplaceFieldAccessesWithQueries
- InlineTemp
- ChangePackage
- Rename Method
- IntroduceSingleton
- IntroduceMemento

EncapsulateField is a composite refactoring, as are IntroduceSingleton and IntroduceMemento which introduce design patterns.

In the initial implementation there was a single instance of each Refactoring object. This was soon reconsidered in order to allow for some optimisations. It was also essential in order to allow some information to be collected during the condition checking phase (e.g. the name of a method), and used in the application phase.

4.1.3 Conditions

Conditions are represented by individual Condition objects. These can be composed together using a special ConjunctionCondition class which represents the conjunction of the list of condition objects which it stores. Negation of a condition is represented by a NOTCondition object which stores the condition to be negated. See Figure 4.3.

Initially conjunction of conditions was represented using an ANDCondition class. This represented the conjunction of exactly two conditions. It was found that this made the management of condition objects more difficult than was necessary, as it was not really necessary to represent conjunctions as binary...
Refactoring

- apply(): Change
- getPreCondition(): Condition
- getPostCondition(): Condition

PrimitiveRefactoring

- preCondition: Condition
- postCondition: Condition
- getPreCondition(): Condition
- getPostCondition(): Condition
- apply(): Change

CompositeRefactoring

- refactorings: List
- getPreCondition(): Condition
- getPostCondition(): Condition
- apply(): Change

Condition

- check(): boolean
- holds(): boolean
- conflictsWith(condition: Condition): boolean

ANDCondition

- left: Condition
- right: Condition
- holds(): boolean

ConjunctionCondition

- conditions: List
- holds(): boolean

Figure 4.2: Refactorings

Figure 4.3: Conditions
trees of AND objects. This class was therefore deprecated in favour of the ConjunctionCondition class.

Composite refactorings do not have explicit pre and post conditions. Instead they are calculated according to a particular ConditionAlgorithm - an instance of the Strategy design pattern. There are two types - a precondition algorithm and a postcondition algorithm. Figure 4.4 shows the structure of the strategy classes.

![Condition Algorithm Diagram](image)

The precondition algorithm works by iterating over the individual refactorings in the composite. For each refactoring, the components of the precondition that are not ‘set up’ by the postcondition of the previous refactorings in the composite are added to the overall precondition.

The postcondition algorithm also iterates over the refactorings in the composite. It simply adds all the conditions in the postcondition to the overall postcondition and then resolves any conflict by removing conflicting conditions.

4.1.4 Changes

Changes which the refactoring makes to the code model are represented using Change objects. These are low level changes. The CodeModel provides a method to apply a given change object using the Command pattern. Each Change object must implement a method which returns another Change object to undo the change. Changes follow the composite pattern to allow multiple changes to be represented. See Figure 4.5.

4.1.5 Patterns

It became clear during implementation that there was a need to be able to refer to specific code constructs in the AST for later use, for example when implementing undo. It is not possible to just use AST components as the AST is rebuilt every time the project is parsed, and so the objects become meaningless. Therefore a pattern system was constructed with patterns to represent packages, classes, methods, fields and variables (see Figure 4.6). Two components were required:

Pattern Maker

This is used to make pattern objects from a particular AST component. This makes patterns much easier to use, as it simplifies their creation. For exam-
Figure 4.5: Changes

Figure 4.6: Patterns

4.1.6 Errors and Exceptions

Since there might still be situations during the condition checking and refactoring processes where an error occurs, certain methods throw a type of RefactoringException. An example of this is the BadRefactoringLocationException,
which is thrown if the location pointed to by the user’s cursor is not valid for
the refactoring which is to be applied.

4.2 The Tool Interface

The framework is designed to be independent of the tool which uses it. To
achieve this, the tool must implement a Tool interface and provide various
methods to obtain the current file, the position of the cursor in the file etc. The
division between the tool and the framework is provided by a set of interfaces
and classes which act as a Façade.

The aim of all the components of the tool façade is to allow extension and
overriding as necessary for the particular implementation.

4.2.1 Components

The Tool Interface

All tools must contain a class which implements this interface. The interface is
used by the framework components to communicate with the tool. See Figure
4.7.

The following methods are provided:

- **getToolLocation**
  This returns a Location object indicating the current position of the
  users cursor and the current focused file.

- **getSelectionStartLocation**
  This returns a Location object indicating the position of the start of the
  current selection.

- **getSelectionEndLocation**
  This returns a Location object indicating the position of the end of the
  current selection.
• `addFailure`
  This method is used by the framework to indicate that a particular Condition is a failure (i.e. it did not hold). The tool may then use this information to display information about the failing condition to the user.

• `removeFileRequest`
  The framework calls this method to request a particular file to be removed. The responsibility for this is given to the tool so it can confirm the action with the user if necessary and carry out any additional operations associated with the removal. This might involve, for example, the closing of windows and the removing of files from projects.

• `getUserInput`
  This method returns some String input from the user. It is envisaged that the tool will implement this via some kind of popup dialogue. A message is passed which indicates the nature of the input that is required.

• `displayConfirmationList`
  This method displays a confirmation list to the user and allows the user to modify it. The framework then uses the modified confirmation list to make decisions in the refactoring process.

• `getFileName`
  This method is used by the framework to get a file name for a given class in a given package. The tool will have information such as the directory where source files are stored, and so will be able to construct a valid path name for the new file. It can also prompt the user if necessary.

• `getSelectedPackage`
  This method is used to get the currently selected package for operations which involve packages.

**Change History**

Tools will in general need to represent in some way the history of changes which have been applied, in particular so they can be undone. The Change-History class gives a starting point for such a representation. It uses a tree structure to store nodes which correspond to individual changes. See Figure 4.8.

This is intended as a starting point for tools, which may subclass the Change-History and related classes to store more information or override behaviour. The GUI tool does exactly this.

**Location**

This class represents a generalised location. To accomplish this it stores three pieces of information:

• The file
• The cursor line
• The cursor column
Confirmation List

The ConfirmationList class represents a data structure which consists of string keys and associated boolean values. This is intended to be represented in the tool as a checklist of values to be selected or deselected as the user desires. It is used (for example) to present a list of subclasses to be modified during a pull up method or pull up field refactoring.

Refactorings Factory

This is used by tools to determine what refactorings are available and to obtain Refactoring objects. Tools can subclass the factory in order to offer more or less refactorings. See Figure 4.9.

One modification of the original design was to add RefactoringInfo objects. These are more lightweight than the Refactoring objects themselves and are used to describe refactorings. The objects store information about the name of the refactoring, whether it is composite and if so what the composites are. Each Refactoring object has a single static instance of a RefactoringInfo object to be returned.

Compiler Interface

This is an abstracted interface to a Java compiler which can be used by tools to compile source files. See Figure 4.10. There is currently only one concrete subclass - a wrapper for the javac compiler which ships with Sun’s JDK. Other
The Tool Interface 4. Implementation

subclasses can be implemented for other compilers such as Jikes.

![JavaCompiler class diagram](image)

**Figure 4.10: Java Compiler**

Parser Interface

A ParserFactory is provided, along with an abstraction for the Transmogrify FileParser element which gets the text of a particular piece of Java code from a file on disk. The framework uses the parser factory to choose a Parser object, with the FileParser being the default. Tools can subclass the factory to return their own subclass of the Parser object. See Figure 4.11. This might read program code from, for example, windows in the tool itself, allowing the code to be parsed without necessarily having to save the files. This was the original intention for the GUI Tool, but it was not decided that implementation was not necessary. The reason for this is that in order to test code it would have to be saved and compiled anyway. Also since the undo system is in place undoing changes is not a problem.

![ParserFactory class diagram](image)

**Figure 4.11: Parser Interface**

4.2.2 The Refactoring Process

In order to perform a refactoring, a tool will perform a sequence of operations such as the following:

- Instantiate a CodeModel instance.
- Add files to the CodeModel to be parsed and call the parse() method.
4. Implementation

The Tool Interface

- Instantiate a Refactoring object for the particular factoring to be applied using a RefactoringsFactory instance.

- Obtain the preconditions from the Refactoring and test them with the current CodeModel. The framework will report any conditions which fail by calling the addFailure() method on the tool.

- If the conditions hold, apply the refactoring to the current CodeModel. This results in a Change object. The tool can then choose to apply the change to the code model. It can also undo the change at a later time by calling the getUndoChange() method and applying the resulting change to the code model. The tool can also obtain the postconditions which have been asserted by the application of the refactoring if necessary.

See Figure 4.12.

4.2.3 Handling User Input

User input to the refactoring process is handled in a reasonably simplistic fashion. The tool must provide methods for obtaining the position of the user’s cursor and the coordinates of the current selection. The user can therefore, for example, place their cursor on a method to be renamed or select a block of code to be extracted.

This scheme also gives the tool freedom in terms of interface implementation. The tool does not have to present the user with a text-based editor view of the source - it can be represented in any way as long as the tool can translate user selections into coordinates within the source files. The tool must also provide mechanisms for text based user input, for example when specifying a method name.

4.2.4 Handling Changes and Undo

The framework provides facilities for undoing and redoing changes made by refactorings. However it leaves the management of these changes up to the tool so that they can be represented as desired.

In the simple case, the tool has stored the changes made by a refactoring and can undo it by obtaining the undo changes. These can then be applied to the code model to undo the refactoring. However in some cases the framework cannot store enough information to undo the refactoring by applying changes. In this case it is necessary for the tool to use its own mechanisms to undo the refactoring, for example by storing a textual representation in its own structure.

The tool interface provides an abstract ChangeHistory class which a tool can choose to extend if desired. This class represents the changes in a tree structure and provides methods to add and remove changes. This can be used by the tool to represent a history of the changes that have been made so that the user can choose to undo particular changes.
Figure 4.12: Refactoring Sequence Diagram
Facilities are also provided for handling user edits. These can be represented by the UserEdit change object and dealt with in the same way as any other change.

An important aspect of the undo process is the ability to decide when a refactoring can be undone, and whether it conflicts with any other refactorings. The framework provides facilities to check the validity of an undo operation via its pre and post condition objects. If the post condition of a refactoring still holds it should be safe to undo it. This allows refactorings to be undone out of order. However the checking of these conditions is left up to the tool, so these facilities can be implemented in any way the author of the tool wishes.

4.2.5 File Handling

One other aspect which the tool must handle is file handling. For example, when the user moves a class from one package to another, the tool must actually move the file from one source directory to another. This is handled by methods in the tool interface to deal with ‘file requests’. These requests are generated by the framework during the refactoring process and are dealt with by the tool once refactoring is complete.

4.2.6 Managing Printing

The tool API contains a PrintManager interface. This is a direct extension of the Transmogrify ASTPrintManager. It is used to provide a method to obtain a Writer for a file. When Transmogrify prints the contents of the AST, it calls this method for each file which needs to be printed. Figure 4.13 shows this process.

Tools can use a class which implements this interface to handle printing of the AST and file management themselves.

4.3 Prototype GUI Tool

Work on a prototype GUI tool was begun early in the project. It serves as a test-bed for the framework and tool interface, as well as being a functional refactoring tool.

The tool uses the JEdit text area component to display Java code with syntax highlighting.

Configuration of the tool’s behaviour is achieved via XML configuration files. The Xerces[34] XML parser and the JAXP[35] framework are used to parse these files.

4.3.1 Features

The tool has several features which are designed to help make the refactoring process easier.
Figure 4.13: Printing Sequence Diagram
4. Implementation

Project Files

The user can specify the details of a ‘project’, i.e. a collection of source files, via an XML file. The specified files are then loaded by the tool for parsing. The project file also allows the user to specify compilation attributes such as class paths, and the files that should be opened at startup.

Compilation Testing

The tool allows the user to test that the files in the project all compile correctly before a refactoring is performed. This is an essential feature since one of the assumptions made by the tool is that the code compiles correctly. If it does not then no claims can be made about the safety of refactorings.

Undo

The tool allows the user to undo any refactorings they have performed. User edits are also dealt with. Changes can be viewed in a special frame which shows the change history in a tree structure. Refactorings can be selected from the history to be undone.

Testing of Refactorings

The user can test whether a refactoring can be applied without actually applying the refactoring. This corresponds to the precondition checking phase of the refactoring process.

Change History

The user can see a historical view of the individual changes that have been made to the project. The changes are grouped hierarchically according to the refactoring of which they are a part.

4.3.2 Interface

The tool presents the user with several different panes. Figure 4.14 shows the overall layout.

The refactorings pane (Figure 4.15) shows a tree-structured list of all the refactorings which are available. Composite refactorings are shown with their component refactorings as child nodes in the tree.

The editor panes (Figure 4.16) show a syntax highlighted view of a Java source file. There is one instance of this pane per file.

The package pane (Figure 4.17) shows a package-oriented view of all the classes in the project. It can be used to navigate the class hierarchy and open required files in the editor.

The changes pane shows a tree structured list of the changes that have been made so far. Elements at the top level of the tree represent refactorings that have been applied. These can be expanded to reveal sub-refactorings, which can in turn be expanded to reveal the changes which implement them. Refactorings to be undone can be selected in this frame.
Figure 4.14: Main Window

Figure 4.15: Refactorings Frame
4. Implementation

Prototype GUI Tool

Figure 4.16: Code Frame

Figure 4.17: Package Frame
4.3.3 Architecture

![Diagram showing GUI Tool classes and their relationships]

Figure 4.18: GUI Tool Classes

Interface With Framework

The GUI tool interfaces with the framework using the various classes explained in the previous section, overriding behaviour where necessary. Specifically:

- A file manager has been implemented which implements the PrintManager interface described in the previous section. This stores the contents of files in memory, which allows a decision to be made on whether to save or not after a refactoring has been performed.
- The RefactoringsFactory is used to obtain a list of refactorings and the concrete Refactoring objects associated with them.
- The various file handling functions are implemented by the file manager to allow refactorings such as ChangePackage to work correctly.

Figure 4.19 shows the file manager and its interface with the framework and the rest of the tool.

Configuration

Configuration of the tool is managed through an XML-based configuration file. This specifies properties such as the default project file to use and whether to enable the Java compiler interface or not. Figure 4.20 shows an example configuration file.

Project System

The project system is controlled by an XML-based project file. Some of the ideas in its structure are taken from the structure of an Ant[36] build file. Figure 4.21 shows an example file.

A project file specifies:

- where to find source files
4. Implementation

Prototype GUI Tool

Figure 4.19: Browser Files and Code Frames

```xml
<?xml version='1.0' encoding='utf-8'?>
<config>
  <defaultprojectfile>testcase/project.xml</defaultprojectfile>
  <compiler enabled="true" />
</config>
```

Figure 4.20: An Example Configuration File
Prototype GUI Tool 4. Implementation

```xml
<?xml version='1.0' encoding='utf-8'?>

<project>

  <name>example</name>

  <sourcopath path="src" />
  <classpath path="build" />
  <buildpath path="build" />

  <files>
    <filesset dir="src" />
  </files>

  <initialfiles>
    <file name="src/bank/Bank.java" />
    <file name="src/bank/Account.java" />
    <file name="src/bank/PersonalAccount.java" />
    <file name="src/bank/BusinessAccount.java" />
  </initialfiles>

</project>
```

Figure 4.21: An Example Project File

- where to compile files to
- the class path

The GUI tool uses an instance of the Project class to manage files in the project. The Project class hides the details of XML parsing from the tool. It also provides facilities to save and restore checkpoints when implementing undo. See Figure 4.22.

Compilation

The JavaCompiler interface from the framework is used to provide a method for the user to compile files in the project. All files specified in the project are compiled according to the various settings specified in the project file. See Figure 4.23.

Conditions

The tool allows conditions to be checked via the ‘test’ option, as well of course as checking them before a refactoring is applied. If any conditions fail they are displayed in the failures window (Figure 4.24).

Changes and Undo

The change history is represented by subclassing components provided by tool interface as described in the previous section. This is done mainly to add in-
4. Implementation

Prototype GUI Tool

org.xml.sax.helpers

DefaultHandler

Project

#sourcePath: String
#classPath: String
#buildPath: String
#filename: String
#projectName: String
-fileSet: Set

-parse(): void
+saveCheckpoint(): int
+restoreCheckpoint(checkpointID:int): boolean
+reload(): void

Figure 4.22: Project System

Figure 4.23: Compilation

Figure 4.24: Failures Window
formation about checkpoints (described below) to the history information. See Figure 4.25.

There is a tree-structured change history frame which allows users to view detailed changes and select refactorings to undo. See Figure 4.26.

One problem that needed to be solved was that of non-undoable refactorings. As explained previously, these refactorings cannot be undone directly by the framework. It is therefore necessary to keep some kind of information about the state of the project before the changes were made so that the project can be rolled back.

If we were not to allow undoing of refactorings in an arbitrary order, the tool could simply keep a history of the changes made at each step. Undoing would then be a matter of reverting back to the state before the last change was made.
This would enforce an order on undo operations, but would be quite efficient and easy to implement.

Since we have the ability to undo refactorings in a near arbitrary order via pre and post condition checking, it was desired to give this kind of ability in the tool. For non-undoable refactorings this presents a problem.

The solution taken was reasonably simple. For every non-undoable refactoring that is applied, a 'checkpoint' is stored. This is basically a copy of all the sources in the project. To undo the refactoring, the sources are reverted to the contents of the checkpoint.

This introduces limitations on the order in which undo operations can be performed. Specifically, one cannot undo a refactoring that was applied before a particular non-undoable refactoring. This is because the checkpoint for the non-undoable refactoring stores the state when the refactoring to be undone had been applied, so restoring to that checkpoint at a later time would be invalid. Also when a non-undoable refactoring is undone, all changes that follow it in the change history are removed. This is because reverting to the checkpoint gives a set of sources as if the proceeding changes never occurred.

This solution is deemed reasonable, as it still allows most of the advantages of undoing out of order, since the number of non-undoable refactorings is small.

A more efficient method of storing the checkpoints could also be devised, for example storing a single checkpoint and then storing the rest as differences from the previous one. The checkpoints could also be stored in compressed form.

User edits are handled in exactly the same way as non-undoable changes. When the user makes a modification to the code, a checkpoint is saved and a user edit added to the change history. The user can then undo the change at a later time, which will involve restoring the checkpoint as described above.

**Code Editor**

Editor windows are implemented as Swing internal frames on a desktop. The text area in each window is an instance of the JEdit text area component. See Figure 4.27. This is a publicly available component which forms part of the JEdit editor. It gives useful features such as syntax highlighting and brace matching.

**Logging and Error Reporting**

One of the main tools used to debug and monitor the operation of the entire framework and tools was logging. This was accomplished using the log4j system. This allows specification of different levels of message priority (e.g. Info, Warning, Error) and customised 'appenders' which handle how the messages are displayed. In the tool, Info and Error level messages are treated as those which the user would need to see, and a custom appender was written which displays messages in a messages window inside the tool (see Figure 4.28). This allows the user to see what is happening during the refactoring process, but hides unnecessary debugging information.
Figure 4.27: Code Frame

Figure 4.28: Messages
4.3.4 Problems Encountered

One significant problem was handling the input and output of source to and from the framework. This involved creating a custom print manager to deal with the contents of the code editor windows, as described above.

Another problem was the implementation of undo functionality as detailed above.

4.4 Emacs Tool

This section describes the implementation of facilities to integrate the framework with the Emacs editor. The framework was integrated with a module for Emacs called the JDEE[38] (Java Development Environment for Emacs). This was chosen for several reasons:

- The JDEE already contains some limited integration with Java programs in order to accomplish code generation tasks, among others.

- It is the author’s development environment of choice, and so work on tool integration would be encouraged as the results would be practically useful in the further development of the project.

The JDEE is primarily written in Emacs Lisp (the extension language on which most Emacs customisation is based). It was therefore necessary to write at least a part of the tool in Lisp.

4.4.1 Interfacing Between Java and Emacs

The construction of the interface between the framework and Emacs can be reduced to two main problems:

Communication between Emacs and the framework

This occurs, for example, when the user wishes to initiate a refactoring. This requires methods to be called in the framework to obtain the refactoring, check preconditions and apply changes.

The JDEE uses another project called the BeanShell[39] to make calls on Java code from Emacs Lisp. This allows a Lisp function to be created which will call a static Java method. The Java method can return Lisp terms to the function via the standard output of the Java process. The JDEE provides such a function which will execute the method call in a given string and evaluate the resulting lisp terms.

Communication between the framework and Emacs

This occurs during the refactoring process. At any point the framework may make calls on the tool to determine various things, for example to prompt the user to input a method name, or to find the current position of the cursor.
The method chosen to implement this communication is based on the Gnuserv system\[40\]. Gnuserv allows one to run a single instance of Emacs and use a client program (gnuclient) to connect to it. The instance of Emacs in use might be on a remote machine.

In normal use the gnuclient program acts exactly like a fully interactive instance of Emacs. The feature of interest for this project is that it also has a command line batch mode which allows an Emacs Lisp form to be evaluated on the command line. This functionality is exactly what is desired, as it is then simple to execute Lisp functions to, say, prompt for input or get the cursor position.

There were two options for the implementation at this point:

1. Communicate directly with the gnuserv process by opening a TCP socket with it from within the Java code and emulating the gnuclient protocol.
2. Call the gnuclient program as a separate process using the Java native execution facilities.

The first option initially seemed the most attractive, as it removed any portability problems caused by having to execute an external program. In fact a basic system was implemented using this method, but problems were uncovered. The main one was that versions of gnuclient on different platforms use different methods of authentication (for example the Unix version uses an X-Windows authentication protocol). These would be difficult to emulate in Java and would impact the portability of the code.

It was therefore decided that the second approach would be adopted. Despite the fact that the overhead of starting a separate process for each call would be higher than the socket approach, this was deemed negligible as it would be outweighed by the complexity of the ongoing refactoring computation.

4.4.2 Architecture

The following diagram (Figure 4.29) shows the proposed structure for the tool.

![Figure 4.29: Emacs Tool Architecture](image)

The implementation therefore has two main components - a Java component and an Emacs Lisp component.
4. Implementation

Tool Interface Implementation

Each method from the Tool interface makes calls on Emacs via the GNUServInterface object (see Figure 4.30) and collates the results into a form appropriate for the framework. For example the getToolLocation() method (which obtains the current location of the cursor) evaluates an Emacs Lisp function which results in a Lisp term indicating the current file, line and column. The Java code parses this and creates a Location object, which it then passes back to the framework.

Regular expressions are used to extract information from the Lisp forms.

Interface With GnuClient/GnuDoit

A Factory class is used to create a GNUServInterface compatible object, depending on operating system in use and availability of programs. On Unix systems the gnuclient program is used in preference, while under Windows the gnudoit program must be used.

Since Emacs (using the Beanshell) can only evaluate static method calls, all methods need to be accessible from a static context. This was achieved by giving the class an internal static instance member of it’s own type. This is accessed through a getInstance() method. This design is reminiscent of that of the Singleton design pattern, but it does not enforce a singleton instance as the constructor is public. This is intended, as it should be possible to use the tool in a non-static context and create multiple different instances. This might be useful in the future if a more advanced interface between Emacs and Java was developed. Figure 4.31 gives an overview of the main classes involved.

Emacs therefore communicates with the Java code by evaluating expressions such as EmacsTool.getInstance().refactor() via the BeanShell.

Lisp Component

There is also a substantial amount of Lisp code. This falls into two categories. Firstly there is Lisp code to present the refactoring functionality in Emacs (for example adding menu options) and to call the various methods of the tool
via the BeanShell. The other category consists of Lisp functions which will be called by the tool via the Gnuserv interface. These are designed to simplify the results returned to the Java code as much as possible, as it is much easier to do list-based processing in Lisp than in Java.

4.4.3 Problems Encountered

The main problem, as mentioned previously, was finding an effective way of communicating between Emacs and Java and vice-versa.

One other problem that was encountered was that the JDE Java evaluation functions are designed to communicate with the BeanShell process via its standard output. This caused problems with logging information which was also sent via the standard output, as Emacs would attempt to evaluate it as a Lisp expression. Logging could not be removed, as some kind of debugging output was essential. Fortunately the framework uses Log4J which features pluggable methods of outputting logging data. One method can log to a server listening on a remote TCP socket. This method was used to obtain logging information without disrupting evaluation.

4.5 XML Refactoring Description

Using XML to describe refactorings was an extension to the project which was conceived early on in the project, but it was not expected that any implementation work would be done. However towards the end of the project it was realised that, with the rest of the framework now in place, it might be possible to do some work in this area after all.

There are several difficulties which still remain before this feature can be im-
implemented fully, but enough work has been done to enable a complete refactoring, RenameMethod, to be specified in XML and applied using the framework. Chapter 6 gives details of how this work might be taken further.

4.5.1 Architecture

The main additional component is the XMLRefactoring class, which is a direct subclass of the Refactoring class (see Figure 4.33). An XML refactoring object reads an XML file and extracts a description of a refactoring from it. The class uses the DOM (Document Object Model) interface to the XML file to extract the document structure

Appendix E gives examples of XML descriptions. An example for the RenameMethod refactoring is given in section E.1. A DTD (Document Type Definition) has been created for refactorings described in the framework. It can be seen in section E.2.

When the XML for a refactoring is parsed, the validation features of the parser are used to ensure that the file conforms to the DTD. This improves error handling and allows the user to use DTD-aware editors, making the process of writing the refactoring descriptions simpler and less error-prone.

A single XML file can specify several refactorings, however each must be given a unique name. A single XMLRefactoring object represents a single refactoring with a given name.

**XML Refactorings Factory**

![XML Refactorings Factory Diagram](image)

To the framework, an XML refactoring behaves in exactly the same way as a normal refactoring, presenting methods to get pre and post conditions and to apply the refactoring. The only difference comes in the way the refactorings are instantiated. This is handled transparently by the XMLRefactoringsFactory class (see figure 4.32), a type of RefactoringsFactory. The tool can use an instance of this class to list and instantiate available XML refactoring objects without any knowledge of how they are represented. The XML refactorings factory parses the XML file on a simple level to determine enough characteristics to return to the tool.
Value Database

The XML specification allows constructs which correspond to creation of values with given identifiers. For example, a value might be a pattern object created in the precursor, or the result of an analysis function (see below). The value database stores these values so that they can be referred to later. It provides simple methods to add and access values given a particular identifier.

Instance Maker

A major task that needs to be performed during the processing of an XML refactoring description is the instantiation of objects for the various conditions, changes and analysis functions specified. The instance maker encapsulates functionality to instantiate these objects given the relevant parameters.

Analysis Functions

Analysis functions are used to specify complex analysis which is done by the refactoring, but which is too complex to be specified in XML (at least with the current implementation). It is intended that there should be a library of these analysis functions (which are in fact each represented by a separate class), in much the same way that there is a library of changes.

4.5.2 The XML Description

The XML description of a refactoring consists of four sections - the precursor, the precondition, the postcondition and the changes.

Precursor

This specifies the information needed to describe the refactoring. Elements declared here are used in the proceeding sections.

The following elements can be used in this section:

- **Target**
  The `target` tag is used to refer to the definition which is the target of the
refactoring and assert it as a value in the value database with the given valueid.

- **User Input**
The `userinput` tag obtains string input from the user and stores it as a value with the identifier specified by valueid.

- **Pattern**
The `pattern` tag obtains a pattern instance of a particular type and stores it as a value with the identifier specified by valueid. This pattern object is usually created by specifying a `template`, which is another pattern object which the pattern is based on. It can also be created from the definition which is the target of the refactoring via the `def` parameter. Other parts of the pattern (for example a method name), are specified as sub elements of the pattern element. If a template or definition is specified, these values override the values obtained from the template or definition.

- **Analysis**
The `analysis` tag obtains the result of an analysis function with a given name, and stores it as a value with a given valueid. Parameters for the analysis function are specified by nested `param` tags.

  `param` tags are specified with a `name`, and contain a tag representing the value of the parameter, for example a pattern or condition. If the parameter to be passed is a list, then a nested `list` element is used.

**Precondition**
This section specifies the precondition for the refactoring. Condition elements may appear here. Conditions may be composite, i.e. contain other conditions. Each condition specified corresponds directly to the creation of the relevant condition object.

Condition are specified with a `name` attribute. Parameters are specified by nested `param` tags.

**PostCondition**
This section specifies the postcondition for the refactoring in a similar way to the precondition.

**Changes**
This section specifies the changes that are made by the refactoring

Changes are specified by `change` tags, with a given `name`. Each change specified corresponds to the creation of a change object. Parameters are specified with nested `param` tags.

An `iterate` tag is also valid here, allowing multiple changes to be specified. The `iterate` tag evaluates its body multiple times, according to the number of elements of the list specified by the `listid` attribute. On each iteration, the corresponding value of the list is asserted in the value database with the id specified by valueid.
4.5.3 Problems Encountered

The main problems that were encountered involved decisions about the structure of the XML description. It was difficult to strike a balance between simplicity and versatility. There are still several problems to be overcome in this area, and these are discussed in Chapter 6.

4.6 Libraries

The project uses many different freely available libraries.

- **Transmogrify**
  This is the main library used in the project. It is used to produce and analyse the AST. See the background section for more information.

- **JEdit**
  This is used to generate syntax-highlighted text areas to allow the user to view and edit Java source files.

- **Xerces**
  This is used to parse the various XML files which are used to configure the tool, and also in the XML refactoring component.

- **log4j**
  This is used throughout the framework for logging and debugging. The tool also uses it to provide log information to the user via a customised appender.

- **JUnit**
  This is used to unit test the framework. Tests are run every time framework code is compiled.

- **ORO**
  ORO is a regular expression library for Java. This is used at many points in the project, especially in the Emacs tool where regular expressions are a useful way of parsing Lisp expressions in Java.

4.7 Summary

A framework has been created based on the design given in the previous chapter. Several new components have been added in order to accommodate new functionality which was required as implementation progressed, for example the pattern matching system. Elements have also been added to the tool interface to improve the range of interaction which can be supported.

An interactive tool has been created as intended. It is a usable refactoring tool and editor. Several features have been added to the tool to make it more convenient to use, for example the ability to group code into projects and to check that code compiles before refactoring.
4. Implementation

Summary

It has also been possible to implement extensions to the project. Refactorings which introduce design patterns have been implemented, as well as an interface to the Emacs tool and the ability to describe some refactorings in XML.

The implementation itself was challenging from a software engineering point of view. Significant work was required to produce a design which broke the complicated process down into simple steps, and producing the corresponding program code was a sizeable task.
Chapter 5

Testing

In order to test that the framework behaves in the way it was specified to, various testing methods have been employed. For the framework itself, unit tests have been used. The tools (the prototype GUI tool and the Emacs tool) have been mostly tested in an interactive fashion. Testing the tools also corresponds to testing the framework, as the tools exercise all parts of the framework in their operation.

5.1 Testing Strategy

The JUnit unit testing framework[6] has been used throughout the project. The concept of using unit testing to drive design, as discussed by Beck[5] and Fowler[4], has been employed to a limited extent.

Functional testing has been performed by testing the refactorings that have been implemented with known situations where they can be performed. In some cases these examples have been drawn from the refactoring literature[4, 17], in other cases the examples have been constructed specifically for this project.

5.2 Unit Testing

Unit tests have been constructed for all major components of the framework using the JUnit framework.

5.2.1 Examples

Some example unit tests have been given here to illustrate how unit testing is performed.

Conditions

The tests given here are simplified examples of basic unit tests which are performed on Condition objects.
The code in Figure 5.1 checks the basic property that a TrueCondition will always hold and a FalseCondition will not.

```java
public void testTrueCondition() {
    Condition condition = new TrueCondition();
    assertTrue(condition.holds(codeModel, tool));
}

public void testFalseCondition() {
    Condition condition = new FalseCondition();
    assertTrue(!condition.holds(codeModel, tool));
}
```

Figure 5.1: Simple Condition Unit Test

Figure 5.2 shows a slightly more complicated test. This code tests the ConjunctionCondition class, which represents the conjunction of several conditions. There are in fact more assertions made about different combinations of conditions in the actual test, but they have been omitted for brevity.

```java
public void testConjunctionCondition() {
    List conditions = new ArrayList();
    conditions.add(new TrueCondition());
    conditions.add(new TrueCondition());
    conditions.add(new TrueCondition());
    Condition conjunction = new ConjunctionCondition(conditions);
    assertTrue(conjunction.holds(codeModel, tool));

    conditions = new ArrayList();
    conditions.add(new FalseCondition());
    conditions.add(new TrueCondition());
    conditions.add(new TrueCondition());
    conjunction = new ConjunctionCondition(conditions);
    assertTrue(!conjunction.holds(codeModel, tool));
}
```

Figure 5.2: Further Condition Unit Tests

**Code Manipulator**

Figure 5.3 shows some more complicated units tests which are part of the test suite for the CodeManipulator component.

The first test method tests the renameReference method of the CodeManipulator. This is used to rename an identifier indicated by a particular Reference object. The test first creates an identifier node, then creates a reference to it. It renames the reference using the CodeManipulator, and then finally makes an assertion that the text of the node referred to has been changed to the correct value.
5. Testing

Unit Testing

public void testRenameReference() {
    String initialText = "test";
    String renamedText = "renamed";

    // make a test node
    SymTabAST node = codeModel.getCodeMaker().makeIdent(initialText);

    // make a reference to the node
    Reference ref = new Reference(node);

    // rename the reference
    codeManipulator.renameReference(ref, renamedText);

    // the node should now be renamed
    assertEquals(renamedText, node.getText());
}

public void testUpdateFile() {
    // load a test file
    File file = new File("testcode/PublicField.java");
    assertNotNull(file);
    codeModel.addFileToParse(file);
    codeModel.parse();

    // get the AST node for the file
    SymTabAST fileNode = codeManipulator.getFileNode(file);
    assertNotNull(file);

    // all nodes of the file node are in the test file
    assertTrue(allNodesInFile(fileNode, file));

    // create a temp file
    createTempFile();
    assertNotNull(tempFile);

    // update the AST nodes for the new file
    codeManipulator.updateFile(fileNode, tempFile);

    // all nodes are are in the new file
    assertTrue(allNodesInFile(fileNode, tempFile));
}

Figure 5.3: Code Manipulator Unit Tests
The second test method tests the `updateFile` method of the CodeManipulator. Each node in the AST stores a reference to the file which it is associated with. During some refactoring operations, AST nodes are moved from one file to another and so this reference needs to be changed. The `updateFile` method achieves this. The test first parses a test file to obtain some AST nodes to test and uses an auxiliary method to test that all nodes refer to the correct file. It then creates a temporary file and uses the CodeManipulator to update the nodes with the new file. It then calls the auxiliary method to check that the nodes refer to the new file.

### 5.3 Functional Testing

The process of refactoring code using one of the tools has been used as the main method of testing that the framework functions as expected.

#### 5.3.1 Strategy

The following strategy was used when carrying out functional testing.

- **Obtain test code**
  Code for a particular situation where a refactoring can be applied is either obtained from literature or written.

- **Load the code into the tool**
  The code is loaded into the tool being tested. It is then checked that the code compiles and that the framework can parse the code without error. An error here would, in general, indicate a problem with the code or, in rare cases, the use of constructs which the Transmogrify framework cannot parse correctly.

- **Test the preconditions**
  The preconditions of the refactoring are checked. If the code and the implementation are correct then the precondition will hold. A failure indicates either that the refactoring as described in the framework is not applicable to the code in question, or that there is some error in the implementation of the refactoring in the framework.

- **Apply the refactoring**
  If the preconditions hold, the refactoring is applied. A failure during the refactoring process will, in general, indicate either that there is an error in the implementation of the various changes which make up the refactoring or that the preconditions which have been specified for the refactoring are not complete.

  If the refactoring succeeds, the following properties are verified:

  - Does the resulting code compile and parse? This can be checked from within the tool.
5. Testing

Functional Testing

- Is the resulting code semantically equivalent to what is expected based on what the refactoring is supposed to achieve? This is sometimes difficult to check, and is not something that can easily be automated.

If any of these properties do not hold then either the preconditions are incomplete and the refactoring should not have been applied, or the implementation of the refactoring is incorrect.

- **Undo the refactoring**
  This involves testing whether the postconditions hold. If they do not, then there is either an error in the way the refactoring was implemented, an error in the way the conditions are checked, or the postconditions specified are incomplete.

Once the refactoring has been undone successfully, the following properties are checked:

- Does the resulting code compile and parse? This can be checked from within the tool.
- Is the resulting code semantically equivalent to the initial code before the refactoring was applied?

If any of these properties do not hold then there is either an error in the implementation of the change mechanism or the undo changes for the changes which were initially applied are not implemented correctly. If the refactoring is not undoable by the framework, then a failure indicates a problem with the tool's implementation of undo (for example the checkpoint mechanism).

Testing, applying and undoing a refactoring exercises the following components:

- **Compilation**
  Before the code is refactored it should be compiled. If the tool offers a compile feature then this is exercised here.

- **Refactorings**
  When a refactoring is applied, a Refactoring object associated with the particular refactoring is instantiated. An object is also instantiated for each of the sub-refactorings if the refactoring is composite.

- **Conditions**
  Before a refactoring is applied, its preconditions are tested. This involves instantiation of Condition objects to represent the precondition of the particular refactoring. Condition objects are also created to represent the post condition of the refactoring.

  If the refactoring is composite, then the condition algorithm classes will also be tested. These are used to compute overall preconditions and post conditions for the composite refactoring.
• Changes and Undo  
When the refactoring is applied, Change objects are created to represent the changes which must be made to the code model in order to implement the refactoring.

If the user chooses to undo the refactoring, then undo changes will be requested from the Change objects and these will be applied to the code model. If the change is not undoable by the framework, then the change will be undone by a method supported by the tool (e.g. checkpoints).

• Parsing and Code Generation  
Before conditions can be tested, the code model must parse the various files in the project to produce an AST and symbol table.

After a refactoring has been applied, the code model is then used to print formatted code back to the source files from the modified AST and symbol table.

• User Interface  
Since refactorings are tested interactively, the user interface is exercised. Most components of the interfaces will be used, namely:

  – the display of the code itself, and possibly editing capabilities
  – the method used to select refactorings to apply
  – the display of any condition failures which occur
  – the display of the changes which has been made
  – undo facilities
  – saving and loading of files

5.3.2 An Example Test Case  
Several small projects have been used for testing. The main one is the Bank example shown later in Section 5.4. The code for the framework itself has also been loaded into the tool and refactored in some instances.

The following serves as a small example of a very simple test. It involves the primitive refactoring Rename Method. We use the strategy given above to perform the test, using the prototype GUI tool.

• Obtain test code  
A simple Java class has been created with a method to be renamed. As suggested earlier, the example is taken from Fowler’s refactoring book[4], page 274. The initial version of the class is shown in Figure 5.4.

• Load the code into the tool  
The code loads correctly after a project file has been created for it. The tool can also compile and parse the code without error.

• Test the preconditions  
The precondition (that the new method doesn not exist already) holds.
class Person {
    public String officeAreaCode = "020";
    private String officeNumber = "7438484";

    public String getTelephoneNumber() {
        return "(" + officeAreaCode + "\) " + officeNumber);
    }
}

Figure 5.4: Initial Test Class

class Person {
    public String officeAreaCode = "020";
    private String officeNumber = "7438484";

    public String getOfficeTelephoneNumber() {
        return "(" + officeAreaCode + "\) " + officeNumber);
    }
}

Figure 5.5: Test Class After Refactoring

- **Apply the refactoring**
  Applying the refactoring produces the code shown in Figure 5.5.
  The tool is able to compile and parse the new code. It can be seen easily that the new code is semantically equivalent to the previous code. In the presence of more code, the Person class might have clients. One would have to check that the clients had been modified in a corresponding fashion.

- **Undo the refactoring**
  Undoing the refactoring succeeds, indicating that the post condition for the refactoring holds. The post condition is that the newly renamed method exists, and the old method does not. The code shown in 5.6 is produced.
  The tool can compile and parse the code. It can be seen that the new code is semantically equivalent to the original test code. There are a few minor formatting differences which are to be expected. As before in the presence of other code more tests would have to be made on the clients of the class.

The example given above is very trivial, but serves to illustrate how interactive functional testing is performed in a disciplined way.

The next logical step from this test is to extend the test case to exercise cases
A Refactoring Example

public String getTelephoneNumber() {
    return "(\" + officeAreaCode + \") \" + officeNumber;\n}

Figure 5.6: Test Class After Undo

where the conditions should fail. For example to test the preconditions, one would first create a method in the test class with the same signature as the method being renamed and the same name as that which the method will be renamed to. In this case the precondition would fail as a method with the new name and signature already exists.

To test the postcondition, there are several modifications that can be made once the refactoring has been applied. One is to remove the newly renamed method using the editor capabilities of the tool and then try to undo the refactoring. The postcondition should not hold in this case, and so the refactoring should not be undone. Another possible modification is to add a new method with the original name and the same signature. The postcondition should not hold in this case, and so the again the refactoring should not be undone.

5.4 A Refactoring Example

What follows is an example of how the framework which has been developed might be used to refactor a particular system. It has been used during development to test the system interactively. The prototype GUI tool is used to modify the code.

The system which is being modelled is very simple, but it serves as an example of how one might use the system to refactor larger systems. The model is of a simple bank, where customers may hold personal accounts or business accounts. A customer may make deposits into their account and withdrawals. When a withdrawal is made the bank makes a charge. The overall class structure is shown in Figure 5.7.

We will now look at how refactorings can be applied using the system to improve the design of the code.

5.4.1 Pull Up Field

It can be seen that the two subclasses PersonalAccount and BusinessAccount share the same field (customerName). It would be better if this field was stored in the superclass (Account). We can use the tool to perform this refactoring.
5. Testing

A Refactoring Example

To perform this, the `customerName` field is selected in either the PersonalAccount or BusinessAccount class. The PullUpField refactoring is then selected and applied. The refactoring results in the addition of a new field `customerName` to the Account class and the corresponding fields being removed from the two subclasses.

5.4.2 Pull Up Method

By similar reasoning we might also move the `printBalance` method up from the subclasses into the superclass. This refactoring moves the selected method from the subclass to the superclass and removes any corresponding methods from the subclasses.

5.4.3 Encapsulate Field

It can be seen from Figure 5.7 that the Account class contains a public field `balance`. Object-oriented principles dictate that it would be better to let the Account class manage access to this variable. The encapsulate field composite refactoring can be used to achieve this. This refactoring adds accessor methods for the variable and then changes the visibility of the field so it is not externally visible.

The refactoring is applied by selecting the field to be encapsulated. Two methods are added: `getBalance` and `setBalance`, and the visibility of the field is set to protected. All external references to the field have also been modified to use the accessor methods instead of accessing the field directly. For example:

```java
currentAccount.balance = 1000000;
```
is replaced by:

```java
currentAccount.setBalance(1000000);
```

### 5.4.4 Extract Method

Figure 5.8 shows the `withdraw` method from the `Account` class. Refactoring can be used to simplify this method. Firstly, the method actually performs two functions - it calculates the charge for the withdrawal and then subtracts this and the withdrawal amount (see Figure 5.8). It would be better if the calculation of the charge was in a separate method to better represent the meaning of the code and encourage code reuse.

```java
public void withdraw(float amount) {
    balance -= amount;

    float amountToCharge = 0;
    float normalCharge = 1;
    if(amount < 100) {
        amountToCharge = normalCharge;
    } else {
        amountToCharge = (float)(0.01 * amount);
    }

    balance -= amountToCharge;
}
```

Figure 5.8: The `withdraw` Method

The ExtractMethod refactoring can be used to achieve this. First the block of code which calculates the charge is selected within the tool. Then the refactoring is applied. The tool asks for a name for the new method and the name (`calculateCharge`) is entered. The result is shown in Figure 5.9.

It can be seen that a new method has been added with private visibility which contains the code which was selected. Variables are passed as parameters as necessary and the modified variable is returned.

### 5.4.5 InlineTemp

The newly extracted method can be further simplified. Referring to Figure 5.9 we can see that there is a temporary variable `normalCharge` which is assigned at definition and used only once. We could inline the uses of this variable to make the code more readable.

To apply this refactoring, the variable is selected and the InlineTemp refactoring is applied. This removes the definition of the variable and replaces all uses
public void withdraw(float amount) {
    balance -= amount;
    float amountToCharge = 0;
    amountToCharge = calculateCharge(amountToCharge, amount);
    balance -= amountToCharge;
}

private float calculateCharge(float amountToCharge, float amount) {
    float normalCharge = 1;
    if(amount < 100) {
        amountToCharge = normalCharge;
    } else {
        amountToCharge = (float)(0.01 * amount);
    }
    return amountToCharge;
}

Figure 5.9: The withdraw Method After ExtractMethod

of the variable with the value it was assigned to. The result can be seen in Figure 5.10.

private float calculateCharge(float amountToCharge, float amount) {
    if(amount < 100) {
        amountToCharge = 1;
    } else {
        amountToCharge = (float)(0.01 * amount);
    }
    return amountToCharge;
}

Figure 5.10: The calculateCharge Method After InlineTemp

5.4.6 IntroduceSingleton

The final example refactoring demonstrated is IntroduceSingleton. This is a Design Pattern refactoring which introduces the Singleton[2] pattern to a class.

The Singleton pattern ensures that there is only a single instance of the class available. It might be reasonable to enforce this condition on the Bank class, as there should only be one bank in the model. Figure 5.11 shows the Bank class before refactoring. The other methods have been removed for brevity.

To apply the refactoring, the Bank class is selected in the tool and IntroduceSingleton is applied. The results can be seen in Figure 5.12. Again other methods have been removed.
package bank;

import java.util.*;

public class Bank {
    private Collection accounts;

    public Bank() {
        accounts = new ArrayList();
    }

    /*
     * other methods
     */
}

Figure 5.11: The Bank Class Before IntroduceSingleton

package bank;

import java.util.*;

public class Bank {
    private static Bank instance;
    private Collection accounts;

    protected Bank() {
        accounts = new ArrayList();
    }

    public static Bank getInstance() {
        if(instance == null) {
            instance = new Bank();
        }
        return instance;
    }

    /*
     * other methods
     */
}

Figure 5.12: The Bank Class After IntroduceSingleton
As can be seen, the visibility of the constructor has been made protected. This ensures that the Bank class and its subclasses control the creation of Bank objects. A static method `getInstance` has been added which returns the value of the static member `instance`, instantiating a Bank object if necessary. All instantiations of Bank objects have been replaced by calls to the `getInstance` method, for example:

```java
Bank bank = new Bank();
```

is replaced with:

```java
Bank bank = Bank.getInstance();
```

### 5.5 Summary

The system has been tested by unit testing during development and by functional testing in an interactive fashion. Much test data has been created with which to test the framework, including a simple real world example which demonstrates the usefulness of the tool.
Chapter 6

Conclusions

The aim of this chapter is to evaluate the key achievements and limitations of the work undertaken, and to arrive at some conclusion about the overall worth of the project.

6.1 Evaluation

The following criteria were specified at the outset to define successful completion of the project:

A framework for refactoring must have been created. It must have the following properties:

- Simplicity of addition for new refactorings.
- Support for both primitive and composite refactorings, and the implementation of several primitive refactorings as a proof of concept.
- Independence of development environment.
- A usable interface in the form of a Refactoring Code Browser.

It can be said that these goals have been broadly achieved. The framework API gives substantial facilities for the implementation of new refactorings. The refactorings which have been implemented using the API thus far demonstrate that it is feasible to use the framework to create new refactorings.

Both primitive and composite refactorings are supported, and examples of both types have been implemented.

It can be seen that the framework that has been implemented is well separated from the development environment in which it is used. This has been one of the fundamental design goals from the outset and is demonstrated by the fact that the tool has been integrated into two separate environments (namely the prototype GUI tool and Emacs). The prototype GUI tool is the manifestation of a usable refactoring code browser.

At the initial stages of the project some consideration was given to a possible extension that could be made to the project. The following possible extensions were considered:
• Implementation of refactoring to design patterns.
• Implementation of user-driven refactoring composition.
• Usable integration with a pre-existing development environment.
• Automatic determination of refactorings that could be applied to a given piece of code.
• XML description of refactorings.

In fact it was possible to implement three of these, namely the implementation of refactoring to design patterns, the integration of the tool into another development environment: Emacs, and XML descriptions of refactorings. Some thought has also been given to other options - this can be seen in Chapter 6.

It is envisaged that the design of the framework in its current state does not prohibit the implementation of the other possible extensions - the main limiting factor has been time.

6.1.1 Analysis

It is worthwhile at this point to make some kind of analysis with regards to the performance of the system that has been created.

One type of analysis that might be done is a formal analysis of the correctness of the refactorings presented. Some initial work has been done on this (see Appendix B). Although not in any way a proof of correctness, the logical formulae help to explain the operation of the refactoring in an unambiguous way. In several cases this highlighted situations where pre and post conditions were incomplete and needed to be improved. A strict formal analysis of refactorings might be a subject for further work.

The nature of the system makes it somewhat difficult to do any kind of quantitative evaluation of the framework. One option might be to compare performance with other refactoring tools. However, it is very difficult to make a fair comparison as the individual refactorings are often implemented in quite diverse ways. Also, the environment in which the refactoring tools operate is often very different, for example one might compare a stand-alone tool with one integrated into a large development environment.

It is however feasible to make some kind of qualitative evaluation. One requirement that was set out at the beginning was that it should be ‘easier’ to use the tool to perform refactorings than to perform them manually. It is reasonable to suggest that this is true in all cases where the relevant refactoring has been implemented in the framework. Using the prototype GUI tool it is a relatively simple matter to load a project and perform a refactoring on it. Although the automated way in which the code is modified is the main benefit, another great advantage over a manual approach is the checking of safety conditions. When refactoring manually it is very easy to make mistakes with stage.

Some qualitative comparisons can also be made with other refactoring tools. From the user’s point of view, many of the features will appear similar to most other refactoring tools. However, there are some features which stand out. One of these is the ability to undo refactorings in a way which does not necessarily
enforce a chronological ordering on the undo process. Another is the explicit modelling of pre and post conditions of the refactorings. This lets the user see exactly what the conditions are and understand better why a refactoring cannot be applied. The explicit modelling of individual changes is also useful to allow the user to get an insight into what each refactoring actually involves. The user might also gain some benefit from the tool independence feature as it may mean there is no need to change development environments to start using the framework.

The work that has been done on XML refactoring description also stands out. This feature is original to this project, and has been made possible mainly by the generic design of the framework itself. Although there is still work to be done in this area, what has been implemented so far shows it to be a useful feature and it is possible that eventually all refactorings in the framework might be described in this way.

One weakness in terms of other refactoring systems might be the catalogue of refactorings available. The number of refactorings presented here - while it compares favourably with that found in most academic works on refactoring - is lower than that found in some commercial tools. It should nonetheless be straightforward to add more refactorings to the catalogue in the future. It could also be said that some of the refactorings that have been implemented are more interesting from a technical point of view, for example in the case of composite and design pattern refactorings.

It is useful to try to identify the strengths and weaknesses of the system as it has been implemented. The main strengths lie in the framework that has been created. It is tool-independent, a feature which few refactoring systems boast, and attempts to foster a particular structure in the refactorings which have been implemented. Specifically it enforces explicit modelling of conditions - both pre and post - and also of changes. It is hoped that this encourages correctness, reuse and simplicity, and one might reasonably say this is the case after examining the refactorings which have been implemented.

Another strength of the framework is the representation of composite refactorings. This has many benefits, including increased generality, reuse and the possibility to add features such as user-driven composition in the future. Composite refactorings have also lead to the implementation of another feature: the ability to introduce design patterns to code via refactoring. This feature is not widely available in other tools, and in the opinion of the author is definitely very useful.

As mentioned previously, another strength is the ability to undo refactorings out of order. Additionally one can also undo individual refactorings which are part of a composite refactoring.

XML refactoring description is another strength of the framework which must be identified. It is a novel feature which is seemingly unique to this project.

However, there are also some weaknesses that can be identified which might be potential targets for future improvement. One is that the explicit modelling of conditions and changes makes the program structure of the framework more complicated than more simplistic systems. This enforced separation might also lead to some performance penalty in some cases, although this has not been
observed thus far. Nevertheless, it can be said that the advantages of this approach as described previously significantly outweigh this disadvantage.

Another weakness of the design might lie in the tool interface. Although it is more than general enough to support most types of editors and integrated development environments, it is still very much based on very editor-centric information such as explicit column and line numbers to get information from the tool. This type of interface might prove limiting if one considers higher level CASE tools which work very much on the design level. For these systems it would be favourable to model the interface on the program entity level rather than the implementation level. That is not to say that this kind of interface is not possible with some amount of restructuring of the current design.

6.2 Contributions

The following is a summary of what the author perceives are the main contributions of this work.

- **A more structured approach to refactoring**
  The constructs which make up the refactoring framework are designed so that they enforce a certain structure on the refactorings that are implemented. See Chapter 3 for details.

- **A tool independent framework**
  The framework that has been designed and implemented in this project is genuinely independent of the tool in which it used.

- **Proof of concept for composite refactorings**
  Composite refactorings have been implemented, and the framework provides a basis on which more can easily be built.

- **Proof of concept for refactorings which introduce design patterns**
  These types of refactorings have been implemented as composite refactorings within the framework.

- **A reusable framework for code manipulation**
  The project has built on the Transmogrify framework to provide extended features for modifying Java code. This has been used in other projects\[41\] to provide code manipulation functionality.

- **Two usable refactoring tools**
  The project has created two refactoring tools - namely the GUI tool which was developed specifically for the project, and a port of the framework to the Java Development Environment for Emacs.

- **XML Description of Refactorings**
  The framework has features which allow refactorings to be specified in XML - a novel feature which has much potential for further development.
6.3 Further Work

Although the implementation of this project has been broadly successful, there are some areas which might be improved upon and many areas in which further work could be undertaken.

In terms of improvements to the current design, the obvious route to take with regard to future work is to try to improve the catalogue of refactorings which it currently supports. The framework has enough infrastructure in place to allow most refactorings to be implemented in a straightforward fashion. It would also be beneficial to investigate further into composite refactorings and design pattern functionality.

As explained in the evaluation section previously, it might also be useful to work on making the interface to the framework more general to accommodate more high level tools. The framework could of course also be integrated into other tools and development environments.

Some work has been done on some of the possible ways in which the project might be extended:

6.3.1 Automated Application of Refactorings

One area which might be explored is that of automated refactoring application. That is producing a tool which will attempt to highlight areas in code where refactorings could be applied, and possibly automatically apply them. The interface to the framework is general enough to be enable refactorings to be applied without user intervention, so the problem which remains is to determine where to apply them. There are several techniques which could be applied to solve this problem, such as Code Smell Analysis and techniques to find duplicate code (see the background section for more information). There is also the work on anti-patterns (patterns in code that indicate bad practice) to be considered.

There are two components to this problem:

- Determining what refactorings might be applied and where
- Applying the given refactorings automatically.

The first problem has been researched to quite a large extent as described in the background section. There are many potential difficulties, including:

- How to determine where a refactoring is necessary. This could be done using Code Smell Analysis, or by looking for anti-patterns.
- How to determine the order in which refactorings should be applied. If a refactoring is performed, it might preclude the application of other more useful refactorings which might otherwise be applied.

Most of the problem of automatic application of a particular refactoring is solved by the framework presented here. Although refactorings are usually
applied in an interactive fashion, the framework lends itself to automation. Therefore once it has been determined what refactorings to apply and where to apply them it should be relatively simple to apply the refactorings using the framework.

### 6.3.2 User-Driven Refactoring Composition

Given the ability to represent refactorings which are composites of other refactorings, it might be possible to allow the user to create composite refactorings themselves in an interactive fashion.

For this to be viable, the framework must have the following features:

- Computation of overall pre and post conditions. This is already available in the current framework.
- Some method of passing analysis information between the various component refactorings and specifying the high-level behaviour of the refactoring.
- Some kind of interactive interface allowing users to compose refactorings together.

The second item is the most difficult problem to solve. In the current framework, even though the various components of the refactoring (for example conditions and changes) are well separated, there is still significant independent processing done by the refactoring to determine what conditions and changes are necessary and the values of the various parameters involved. It would be difficult to give the user some kind of interface to control what processing takes place at this stage.

One way to solve this problem might be to create some kind of high level language which would allow the user to specify this behaviour in a general way. The first steps towards this type of system have been taken in the work on XML descriptions of refactorings. However, this strategy is still not suited to interactive use.

To make this kind of system worthwhile, one would also need a large base of refactorings for the user to choose from.

### 6.3.3 XML Description of Refactorings

As explained in Chapter 4, some work has been done on describing refactorings within the framework in XML. There is however much work still to do in this area.

The main problem is the creation of a framework where the action of a refactoring can be specified with enough granularity. At the current level of implementation it is possible to specify simple refactorings such as RenameMethod which involve only simple processing to create the relevant instances of the condition and change classes. For more complicated refactorings the system is
6. Conclusions

Summary

not rich enough to translate the processing done in the refactoring class directly to an XML description.

In terms of further work, one main feature which must be implemented is support for the description of composite refactorings. Another aspect which needs to be improved is the value database. When values are created and referenced there should be more type and consistency checking and more detailed reporting of inconsistencies to the user.

The main efforts of further work should be concentrated on enriching the syntax of XML descriptions so that it is both simpler to represent refactorings and also possible to represent the more complicated refactorings which cannot currently be supported. Structures should be introduced that allow analysis information to be processed and propagated between elements in a simple and generic way.

6.3.4 Extensions for J2EE

As identified in Chapter 3, special consideration must be taken when refactoring Java programs which are part of J2EE (Java 2 Enterprise Edition) applications. A possible extension for the project might be to incorporate support for these applications.

This would involve adding functionality to the framework so that it is aware of the various J2EE constructs and the constraints which are placed on them. For example, when refactoring a class which is an Enterprise JavaBean there are many restrictions which have to be obeyed. Some language features are forbidden which would be valid in a normal context. The tool would therefore have to obey these restrictions.

The tool would also have to be aware of the XML-based deployment descriptors which are an important part of a J2EE application. The contents of these files puts constraints on the contents of the corresponding classes. For example in an Entity Bean, the XML descriptor specifies the fields of the class which are persisted. This results in, among other things, accessor (get and set) methods being implicitly created for the field. It would therefore be incorrect for a refactoring to create explicit get and set methods overriding those implemented by the container.

In spite of these difficulties, several refactorings have been proposed for J2EE applications, and design pattern work has also emerged[42]. This is therefore definitely a promising area for further work.

6.4 Summary

In conclusion, it can reasonably be said that this project has achieved its initial aims - namely to create a refactoring framework that is independent of the development environment in which it is used, and to create a prototype tool which uses the framework. In fact, the project has been extended further than was initially anticipated.
The project has given validity to the concepts of composite refactorings and refactoring to introduce design patterns. It has also shown that it is possible to create a refactoring framework that is independent of the tool in which it is used.

In the opinion of the author, this report demonstrates that the structured approach to implementation of refactorings presented is viable, and is an approach that should be adopted by refactoring systems in order to obtain the benefits that have been outlined here.

The project has resulted in the creation of two useful pieces of software - a refactoring framework which is both general and extensible, and a well-featured refactoring tool.

Generally speaking, this project provides more motivation to demonstrate that refactoring is a valid and useful method in software design and implementation, and that tool support is an essential component in the refactoring process.
List of Figures

3.1 Framework Overview ........................................... 27
3.2 Overview .................................................. 28
3.3 Refactoring .................................................. 29
3.4 Model of Conditions .......................................... 29
3.5 Condition .................................................... 30
3.6 Code Model ................................................... 31
3.7 Model of Changes ............................................ 32
3.8 Tool Interface ................................................ 33
3.9 User Interface ................................................ 34

4.1 Code Model and Related Classes ............................. 36
4.2 Refactorings .................................................. 38
4.3 Conditions .................................................... 38
4.4 Condition Algorithm ......................................... 39
4.5 Changes ....................................................... 40
4.6 Patterns ....................................................... 40
4.7 Tool Interface ................................................ 41
4.8 Change History ............................................... 43
4.9 Refactorings Factory ......................................... 43
4.10 Java Compiler ............................................... 44
4.11 Parser Interface ............................................. 44
4.12 Refactoring Sequence Diagram ............................. 46
4.13 Printing Sequence Diagram ................................ 48
4.14 Main Window ............................................... 50
4.15 Refactorings Frame ......................................... 50
4.16 Code Frame .................................................. 51
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.17</td>
<td>Package Frame</td>
<td>51</td>
</tr>
<tr>
<td>4.18</td>
<td>GUI Tool Classes</td>
<td>52</td>
</tr>
<tr>
<td>4.19</td>
<td>Browser Files and Code Frames</td>
<td>53</td>
</tr>
<tr>
<td>4.20</td>
<td>An Example Configuration File</td>
<td>53</td>
</tr>
<tr>
<td>4.21</td>
<td>An Example Project File</td>
<td>54</td>
</tr>
<tr>
<td>4.22</td>
<td>Project System</td>
<td>55</td>
</tr>
<tr>
<td>4.23</td>
<td>Compilation</td>
<td>55</td>
</tr>
<tr>
<td>4.24</td>
<td>Failures Window</td>
<td>55</td>
</tr>
<tr>
<td>4.25</td>
<td>Changes Model</td>
<td>56</td>
</tr>
<tr>
<td>4.26</td>
<td>Changes Frame</td>
<td>56</td>
</tr>
<tr>
<td>4.27</td>
<td>Code Frame</td>
<td>58</td>
</tr>
<tr>
<td>4.28</td>
<td>Messages</td>
<td>58</td>
</tr>
<tr>
<td>4.29</td>
<td>Emacs Tool Architecture</td>
<td>60</td>
</tr>
<tr>
<td>4.30</td>
<td>Gnuserv Interface</td>
<td>61</td>
</tr>
<tr>
<td>4.31</td>
<td>Emacs Tool</td>
<td>62</td>
</tr>
<tr>
<td>4.32</td>
<td>XML Refactorings Factory</td>
<td>63</td>
</tr>
<tr>
<td>4.33</td>
<td>XML Refactoring</td>
<td>64</td>
</tr>
<tr>
<td>5.1</td>
<td>Simple Condition Unit Test</td>
<td>70</td>
</tr>
<tr>
<td>5.2</td>
<td>Further Condition Unit Tests</td>
<td>70</td>
</tr>
<tr>
<td>5.3</td>
<td>Code Manipulator Unit Tests</td>
<td>71</td>
</tr>
<tr>
<td>5.4</td>
<td>Initial Test Class</td>
<td>75</td>
</tr>
<tr>
<td>5.5</td>
<td>Test Class After Refactoring</td>
<td>75</td>
</tr>
<tr>
<td>5.6</td>
<td>Test Class After Undo</td>
<td>76</td>
</tr>
<tr>
<td>5.7</td>
<td>The Bank Example</td>
<td>77</td>
</tr>
<tr>
<td>5.8</td>
<td>The withdraw Method</td>
<td>78</td>
</tr>
<tr>
<td>5.9</td>
<td>The withdraw Method After ExtractMethod</td>
<td>79</td>
</tr>
<tr>
<td>5.10</td>
<td>The calculateCharge Method After InlineTemp</td>
<td>79</td>
</tr>
<tr>
<td>5.11</td>
<td>The Bank Class Before IntroduceSingleton</td>
<td>80</td>
</tr>
<tr>
<td>5.12</td>
<td>The Bank Class After IntroduceSingleton</td>
<td>80</td>
</tr>
<tr>
<td>C.1</td>
<td>An Example Configuration File</td>
<td>116</td>
</tr>
<tr>
<td>C.2</td>
<td>An Example Project File</td>
<td>116</td>
</tr>
<tr>
<td>C.3</td>
<td>Compilation</td>
<td>118</td>
</tr>
<tr>
<td>C.4</td>
<td>Refactorings Frame</td>
<td>118</td>
</tr>
</tbody>
</table>

92
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.5</td>
<td>Failures Frame</td>
<td>118</td>
</tr>
<tr>
<td>C.6</td>
<td>Changes Frame</td>
<td>119</td>
</tr>
</tbody>
</table>
References


http://groups.yahoo.com/group/refactoring.


[22] Various, “Refactoring browser for java wiki.”

[23] Various, “The eclipse project.”
http://www.eclipse.org/.

http://www.intellij.com/idea.


http://dpt.kupin.de/.


http://www.antlr.org/.

[29] The Sable Researchers Group, “Sablecc.”
   http://www.ittc.ku.edu/JParse/.


   http://java.sun.com/j2ee/.

[33] Free Software Foundation, “GNU general public license.”

[34] Apache Software Foundation, “Xerces XML parser.”

[35] Sun Microsystems, “Java API for XML processing.”
   http://java.sun.com/xml/jaxp/.


[37] Apache Software Foundation, “log4j.”

[38] Paul Kinnucan et al., “The Java Development Environment for Emacs.”
   http://jdee.sunsite.dk.


[40] A. Norman, “The gnuserv system.”
   http://www-uk.hpl.hp.com/people/ange/gnuserv/.


Appendix A

Availability of Sources and Documentation

Sources and documentation for the project are available at the project website:

http://www.freefactor.org

or

http://www.doc.ic.ac.uk/~gjb198/Project

All relevant files are linked to from this page.

A.1 Sources

Source code is available on the following paths relative to the home directory of the user gjb198.

- Freefactor framework:
  Project/freefactor/freefactor-src.tar.gz
- GUI Tool:
  Project/gui-tool/guitool-src.tar.gz
- Emacs Tool:
  Project/emacs-tool/emacstool-src.tar.gz

A.2 Documentation

- This report:
  Project/docs/report/report.tex
  Project/docs/report/report.ps
  Project/docs/report/report.pdf
- Javadoc documentation for the Freefactor API:
  Project/freefactor/docs/api
A.3 Binaries

Compiled distributions of the GUI tool and Emacs tool are also available.

- GUI tool:
  Project/gui-tool/freefactor-guitool.tar.gz

- Emacs tool:
  Project/emacs-tool/freefactor-emacstool.tar.gz
Appendix B

Refactorings

The refactorings which have been implemented in the framework to date are presented here. Some effort has been made to present the refactorings in a formal way, especially in terms of their pre and post conditions.

Each refactoring is presented with a pre and post condition specified in simple logic. Also given is a simple explanation of the action of the refactoring. For composite refactorings, the overall pre and post conditions are computed.

B.1 Notation

Pre and post conditions are expressed in logic in terms of conditions on the code model. We define the sets Package, Type, Method, Variable and Block. The set Type is defined as the union of the sets Class and Interface. We also define some simple auxiliary functions:

- **fields(c)** - the set of fields of class or interface c
- **methods(c)** - the set of methods of class or interface c
- **variables(c)** - the set of variables (fields and locals) in a class c
- **locals(m)** - the set of local variables of method m
- **name(m)** - the name of method m
- **signature(m)** - the signature of method m
- **field(v, c)** - the type of the field v of class or interface c
- **method(m, c)** - the type of the method m of class or interface c
- **type(c, p)** - the class or interface c is defined in package p
- **visibility(i, c)** - the visibility of the identifier i in class c, an element of the set \{public, private, protected, package\}. We also give the ordering \(\geq\) such that \(\text{public} \geq \text{package} \geq \text{protected} \geq \text{private}\), and the other orderings \(<, >, \leq\) with the appropriate meanings.
- \textbf{subtypes}(c) - the set of subtypes of a class or interface c
- \textbf{supertypes}(c) - the set of supertypes of a class or interface c (if c is a class this will have at most one element, but not necessarily if c is an interface).

Other simple auxiliary functions are used where their meaning is obvious from their name.

\section*{B.2 Rename Method}

A method name must be changed to better indicate its meaning or allow other refactorings to be applied.

\textbf{Action}

Change the name of the method and any other implementations in the type hierarchy according to the user’s choice. Any clients of the renamed methods are updated to use the new name.

\textbf{Precursor}

\begin{align*}
C & \subseteq \text{Type}, m \in \text{Method}, \forall c : C[m \in \text{methods}(c)], n \in \text{Name} \\
\text{Precondition} & \\
\forall c : C - \exists m_2 : \text{methods}(c)[\text{signature}(m) = \text{signature}(m_2) \land \text{name}(m_2, n)]
\end{align*}

\textbf{Changes}

- RenameMethod

\textbf{Postcondition}

\begin{align*}
\forall c : C[m \notin \text{methods}(c) \land \\
\exists m_2 : \text{methods}(c)[\text{signature}(m) = \text{signature}(m_2) \land \text{name}(m_2, n)]
\end{align*}

\section*{B.3 Inline Temp}

A variable is assigned only once with a simple expression. Removing the variable would allow other improvements to be made.

\textbf{Action}

Replace all references to the temporary variable with the expression that is assigned to it. Remove the declaration of the variable.
B. Refactorings

Pull Up Field

Precursor

\[ c \in \text{Class}, v \in \text{variables}(c). \]

Precondition

\[ v \notin \text{fields}(c) \land \text{IsAssignedAtDef}(v) \land \neg \text{IsReassigned}(v) \]

Changes

- ReplaceReferences
- RemoveAssignments
- RemoveDefinition

Postcondition

\[ v \notin \text{variables}(c) \]

B.4 Pull Up Field

Two sibling classes have the same field. Move the field up into the superclass.

Action

Move the field declaration up to the superclass and remove from subclasses. The subclasses to be modified are selected by the user. Ensure the visibility is at least protected.

Precursor

\[ c_1, c_2 \in \text{Type}, c_2 \in \text{supertypes}(c_1), v \in \text{fields}(c_1), C \subseteq \text{subtypes}(c_2) \]

Precondition

\[ \text{field}(v, c_1) \neq \text{Udf} \]
\[ \land \text{field}(v, c_2) = \text{Udf} \]
\[ \land \forall c' : C[\text{field}(v, c') \neq \text{Udf} \rightarrow \text{field}(v, c_1) = \text{field}(v, c')] \]

Changes

- RemoveClassMember
- AddClassMember
- AddImportForClass
Postcondition

\[ field(v, c_2) \neq \text{Udf} \land \forall c': C[\text{field}(v, c') = \text{Udf}] \land \text{visibility}(v, c_2) \geq \text{protected} \]

B.5 Pull Up Method

Two sibling classes have the same method. Move the method up into the superclass.

Action

Move the method declaration up to the superclass and remove from the subclasses. The subclasses to be modified are selected by the user. Ensure the visibility is at least protected.

Precursor

\[ c_1, c_2 \in \text{Class}, c_2 \in \text{supertypes}(c_1), m \in \text{methods}(c_1), C \subseteq \text{subtypes}(c_2) \]

Precondition

\[ \text{method}(m, c_1) \neq \text{Udf} \land \text{method}(m, c_2) = \text{Udf} \land \forall c': C[\text{method}(m, c') \neq \text{Udf} \rightarrow \text{method}(m, c_1) = \text{method}(m, c')] \]

Changes

- RemoveMethod
- AddMethod
- AddImportForClass

Postcondition

\[ \text{method}(m, c_1) = \text{Udf} \land \text{method}(m, c_2) \neq \text{Udf} \land \forall c': C[\text{method}(m, c') = \text{Udf}] \land \text{visibility}(v, c_2) \geq \text{protected} \]

B.6 Change Package

A class or interface would be better placed in a different package, to better indicate its function or to allow access to package-private data.
### B. Refactorings

#### Extract Method

**Action**

Change the package declaration in the source file to indicate the new package. Ensure that any references to the class have the correct imports. Remove explicit imports of the old class.

**Precursor**

\[ p_1, p_2 \in \text{Package}, c \in \text{classes}(p_1) \]

**Precondition**

\[
\begin{align*}
\text{class}(c, p_2) \\
\forall f : \text{Field}, c' : \text{Type}[\text{references}(c, f, c') \rightarrow \text{visibility}(f, c') \neq \text{package}] \\
\forall m : \text{Method}, c' : \text{Type}[\text{references}(c, m, c') \rightarrow \text{visibility}(m, c') \neq \text{package}] \\
\forall f : \text{Field}, c' : \text{Type}[\text{references}(c', f, c) \rightarrow \text{visibility}(f, c) \neq \text{package}] \\
\forall m : \text{Method}, c' : \text{Type}[\text{references}(c', m, c) \rightarrow \text{visibility}(m, c) \neq \text{package}]
\end{align*}
\]

**Changes**

- ChangePackage
- AddImports
- RemoveImports
- AddImportForPackage

**Postcondition**

\[
\begin{align*}
\text{type}(c, p_1) \\
\neg \text{type}(c, p_2)
\end{align*}
\]

### B.7 Extract Method

A section of a block of code has a particular purpose. Extract the code into a method to better describe the function of the code and allow it to be reused more easily.

**Action**

Create a method with a descriptive name, the body of which is the code block to be extracted. Pass as parameters any local variables that are used within the block. Return the value of a variable that is assigned within the block, if one exists. Replace the code block with a call to the method with the correct arguments, and assigning the return value to a local variable if appropriate.
Encapsulate Field

B. Refactorings

Precursor
\( c \in \text{Class}, n \in \text{Name}, m \in \text{Method}, \text{name}(m, n), b \in \text{Block}, \text{body}(m, b), v \in \text{locals}(m) \)

Precondition
\[
\begin{align*}
& m \notin \text{methods}(c) \\
& \text{localvar\ modified}(b) \leq 1 \\
& \text{IsAssignedAtDef}(v)
\end{align*}
\]

Changes
- RemoveBlock
- AddMethod
- AddMethodCall

Postcondition
\[
\begin{align*}
& m \in \text{methods}(c) \\
& \text{returns}(m, v)
\end{align*}
\]

B.8 Encapsulate Field

A class has a public field which is accessed externally. Encapsulate the field using accessor methods so that the class can control access.

This is a composite refactoring made up of the following primitive refactorings.

B.8.1 Add Accessors

Add accessor (get and set) methods for the field.

Action
Add a getter method which returns the value of the field and has no arguments.
Add a setter method with an argument of the field type which assigns the argument to the field;

Precursor
\( c \in \text{Class}, f \in \text{variables}(c), m_g \in \text{Method}, \text{IsGetter}(m_g, f), m_s \in \text{Method}, \text{IsSetter}(m_s, f) \)

Precondition
\[
\begin{align*}
& \text{field}(f, c) \\
& m_g \notin \text{methods}(c) \\
& m_s \notin \text{methods}(c)
\end{align*}
\]
B. Refactorings

Changes

- AddMethod

Postcondition

\[ m_2 \in \textit{methods}(c) \land m_s \in \textit{methods}(c) \]

B.8.2 Replace Field Accesses With Queries

Replace accesses to the field with calls to the accessor methods.

Action

Replace all usages of the field with calls to the getter method for the field. Replace all assignments to the field with calls to the setter method.

Precursor

\[ c \in \text{Class}, f \in \text{variables}(c), m_g \in \text{Method}, \text{IsGetter}(m_g, f), m_s \in \text{Method}, \text{IsSetter}(m_s, f) \]

Precondition

\[ \text{field}(f, c) \land m_g \in \textit{methods}(c) \land m_s \in \textit{methods}(c) \]

Changes

- ReplaceFieldAccessesWithQueries

Postcondition

\[ \forall \epsilon : \text{Class}\text{references}(c, f, \epsilon) \rightarrow c = \epsilon' \]

B.8.3 Protect Field

Change the visibility of the field so it is no longer accessible externally.

Action

Change the visibility modifier for the field to protected.

Precursor

\[ c \in \text{Type}, f \in \text{variables}(c) \]

Precondition

\[ \text{field}(f, c) \land \forall \epsilon : \text{Class}\text{references}(c, f, \epsilon) \rightarrow c = \epsilon' \]
**B.9 Introduce Singleton**

Only one instance of a given class should exist. This instance should be globally accessible.

**B.9.1 Add Singleton Instance**

Add a static member to the class to store the singleton instance. Add a static getInstance() method to access the static instance.

**Action**

Create a protected static member instance in the class of the same type as the class. Add a static, public method getInstance with a return type of the class type. The method body for this method returns the value of the static instance member, instantiating it with an instance of the class if it is null (this will occur only once).

**Precursor**

\[ c \in \text{Class}, f \in \text{field.name}(f', \text{instance}'), m \in \text{Method.name}(m', \text{getInstance}') \]
Precondition
\[ field(f, c) = \text{Udf} \]
\[ \land \ \text{method}(m, c) = \text{Udf} \]

Changes
- AddClassMember
- AddMethod

Postcondition
\[ field(f, c) \neq \text{Udf} \]
\[ \land \ \text{method}(m, c) \neq \text{Udf} \]
\[ \land \ \text{visibility}(f) = \text{protected} \]

B.9.2 Replace Instantiations With Singleton Accesses

Replace every instantiation of the class with a call to the getInstance() method.

Action
Replace every instantiation of the class (every `new` call) with a call to the static `getInstance()` method.

Precursor
\[ c \in \text{Class}, f \in \text{field}, \text{name}(f, \text{!} \text{instance}), m \in \text{Method}, \text{name}(m, \text{!} \text{getInstance}) \]

Precondition
\[ field(f, c) \neq \text{Udf} \]
\[ \land \ \text{method}(m, c) \neq \text{Udf} \]

Changes
- ReplaceInstantiationsWithSingletonAccesses

Postcondition
\[ \forall i : \text{instantiations}(c)[\text{contains}(m, i)] \]

B.9.3 Enforce Single Instance

Change the visibility of the constructor of the class to protected. This allows subclasses to override functionality whilst preventing external instantiation.
Action
Change the visibility modifier of the constructor to protected. If no constructor exists, add one with protected access. This ensures only the class or subclasses may instantiate the class.

Precursor
\( c \in \text{Class}, m_c \in \text{Method, constructor}(m_c, c) \)

Precondition
\( \forall c': \text{Class}, i : \text{instantiations}(c)[\text{contains}(c', i) \rightarrow c' \in \text{subclasses}(c)] \)
\( \land \text{parameters}(c) = \emptyset \)

Changes
- AddMethod
- MethodVisibility

Postcondition
\( \text{visibility}(m_c) = \text{protected} \)

Overall

Precursor
\( c \in \text{Class}, f \in \text{field, name}(f', \text{instance'}), m \in \text{Method, name}(m', \text{get Instance'}) \)

Precondition
\( \text{field}(f, c) = \text{Udf} \)
\( \land \text{method}(m, c) = \text{Udf} \)
\( \land \text{parameters}(c) = \emptyset \)

Postcondition
\( \text{field}(f, c) \neq \text{Udf} \)
\( \land \text{method}(m, c) \neq \text{Udf} \)
\( \land \text{visibility}(f) = \text{protected} \)
\( \land \forall i : \text{instantiations}(c)[\text{contains}(m, i)] \)
\( \land \text{visibility}(m_c) = \text{protected} \)

**B.10 Introduce Memento**

Create a class that serves as a memento for another class. Add methods to obtain and restore the memento.
B.10.1 Create Memento Class

Create a class which has fields corresponding to all the fields of the class to act as a memento.

**Action**

Create a class with a name which is the concatenation of the class name and ‘Memento’. Add a corresponding field to this class for each field in the originating class with package-private access.

**Precursor**

\[ c, c' \in \text{Class, name}(c') = \text{name}(c) + \text{Memento}, p \in \text{Package} \]

**Precondition**

\[ \text{type}(c, p) \]
\[ \land \lnot \text{type}(c', p) \]

**Changes**

- AddClass
- AddClassMember
- AddImportForClass

**Postcondition**

\[ \text{type}(c', p) \]
\[ \land \forall f : \text{fields}(c)[\text{field}(f, c) = \text{field}(f, c')] \]

B.10.2 Add Create Memento Operation

Add a method to return a memento instance for the class.

**Action**

Add a method `createMemento` which creates an instance of the corresponding memento object, copies the state of the originator to the fields of the memento and returns the memento.

**Precursor**

\[ c, c' \in \text{Class, name}(c') = \text{name}(c) + \text{Memento}, p \in \text{Package}, m, c \in \text{Method, name}(m, c) = \text{createMemento} \]

**Precondition**

\[ \text{type}(c, p) \]
\[ \land \text{type}(c', p) \]
\[ \land \text{method}(m, c) = \text{Udf} \]
Changes

- AddMethod

Postcondition

\(\text{method}(m, c) \neq \text{Udf}\)

B.10.3 Add Set Memento Operation

Add a method to restore the state of an object from a memento.

Action

Add a method \(\text{setMemento}\) which takes a memento object as a parameter and restores the state of the originator from the fields of the memento.

Precursor

\(c, c' \in \text{Class}, \text{name}(c') = \text{name}(c) \neq \text{Memento}, p \in \text{Package}, m_s \in \text{Method}, \text{name}(m_s) = \text{setMemento}\)

Precondition

\(\text{type}(c, p)\)
\(\wedge \text{type}(c', p)\)
\(\wedge \text{method}(m_s, c) = \text{Udf}\)

Changes

- AddMethod

Postcondition

\(\text{method}(m_s, c) \neq \text{Udf}\)

Overall

Precursor

\(c, c' \in \text{Class}, \text{name}(c') = \text{name}(c) \neq \text{Memento}, p \in \text{Package}, m_c \in \text{Method}, \text{name}(m_c) = \text{createMemento}\)
\(m_s \in \text{Method}, \text{name}(m_s) = \text{setMemento}\)

Precondition

\(\text{class}(c, p)\)
\(\wedge \neg \text{type}(c', p)\)
\(\wedge \text{method}(m_c, c) = \text{Udf}\)
\(\wedge \text{method}(m_s, c) = \text{Udf}\)
Postcondition

\[\text{type}(c', p) \wedge \forall f : \text{fields}(c)[\text{field}(f, c) = \text{field}(f, c')]\wedge \text{method}(m, c) \neq \text{Udef} \wedge \text{method}(m, c) \neq \text{Udef}\]
Appendix C

User Guide

C.1 Prototype GUI Tool

C.1.1 Requirements
The GUI tool requires JDK 1.3 or higher to be installed on your system. The runtime (JRE) is not sufficient if you wish to be able to compile your programs from within the tool. All other libraries are included in the distribution.

C.1.2 Installing and Running the Tool
Obtain the GUI Tool Freefactor distribution from http://www.freefactor.org in either zip or gzipped-tar format. Unpack the archive into an installation directory of your choice.

Before running the tool, ensure that you have the JAVA_HOME variable set to the location of your JDK installation. See the JDK documentation for details.

Windows
Run the freefactor.bat file. This should start the tool. You may wish to put the tool installation directory in your PATH for convenience.

Unix
Execute the freefactor shell script to start the tool. You may wish to put the tool installation directory in your PATH for convenience.

C.1.3 Configuring the tool
The tool is configured by the XML file config.xml in the installation root. An example is given in Figure C.1.

You can use this file to specify a default project file (see below) to load at startup, and to enable or disable the compiler (for example if you only have the JRE installed).
C.1.4 Setting up a project

Projects are collections of source files to be loaded into the tool. The project preferences are controlled by another XML file project.xml in the root of the project directory. There will be one XML file for each project. It is recommended that you arrange each project you work on in its own directory.

An example project is included with the distribution, and the tool is configured to load it at startup. It resides in the example subdirectory of the installation root directory. The project directory has two subdirectories, src where source files are kept and build where class files are compiled to. Figure C.2 shows the project.xml file for this project.

```xml
<?xml version='1.0' encoding='utf-8'?>
<project>
  <name>example</name>
  <sourcepath path="src" />
  <classpath path="build" />
  <buildpath path="build" />
  <files>
    <filesset dir="src" />
  </files>
  <initialfiles>
    <file name="src/bank/Bank.java" />
    <file name="src/bank/Account.java" />
    <file name="src/bank/PersonalAccount.java" />
    <file name="src/bank/BusinessAccount.java" />
  </initialfiles>
</project>
```

Figure C.2: An Example Project File
The file specifies the following properties:

- **Project Name**
  This is just a name used for convenience in the tool

- **Source Path**
  This is the directory where source files are found, relative to the project directory.

- **Build Path**
  This is the directory where class files are compiled to, relative to the project directory.

- **Class Path**
  This is the class path that will be used during compilation of the project. Add any libraries and external classes used by your project here. See the JDK documentation for more details.

- **Files**
  This element specifies the collection of source files that the project contains. Valid elements within the files element are file and fileset. The file element allows you to specify a single file to be included in the source file list via the name attribute. The fileset element allows you to specify a set of files in a particular directory via the dir attribute. All Java source files in the directory and all its subdirectories will be processed.

- **Initial Files**
  This element allows the specification of source files to be opened in the editor at startup. The files are specified in the same way as in the files element above.

### C.1.5 Opening a project

To open a project, or change the current project you are working on, select **Open** from the **Project** menu. A dialogue box will appear allowing you to choose the project.xm1 file of a different project. This will close the current project and open the new one.

### C.1.6 Compiling the project

To compile the files in your project, select **Compile** from the **Project** menu. This will open the compilation window as shown in Figure C.3. The files will be compiled one by one, and a tick will be marked in the check box beside the file if compilation was successful. Once all the files have been compiled the **OK** button will be enabled so that the window can be closed.

It is important that all files are compiled before refactoring can begin, as this is a precondition of safety preservation.
C.1.7 Applying a refactoring

To apply a refactoring, first select the correct program element in one of the code frames and the refactoring from the refactorings frame (Figure C.4). Then click the apply button.

An error will be displayed if the program element selected is not valid for the refactoring concerned. The preconditions for the refactoring will then be checked, if they do not hold then the failures dialogue box will be displayed (Figure C.5). This displays all conditions and whether they hold or not.

C.1.8 Testing a refactoring

A refactoring can be tested without applying it. To do so, proceed in the same way as applying a refactoring but click the test button. This will not apply the
refactoring, but will test its precondition. If any conditions fail, they will be displayed in the failures frame (Figure C.5).

C.1.9 Undoing a refactoring

To undo a refactoring, select it from the changes frame (Figure C.6). The frame displays all changes that have been made. Individual refactorings in a composite refactoring can also be undone.

![Changes Frame](image)

Figure C.6: Changes Frame

It is possible to undo most refactorings in an arbitrary order, because post conditions are checked to ensure the refactoring can be undone. However some refactorings cannot be undone in this way because not enough information can be retrieved from the code. These are displayed in red in the change history. In this case, the tool saves ‘checkpoints’ of the code which it can roll back to. When one of these refactorings is undone, all subsequent refactorings are automatically undone as well. Also it is not possible to undo a refactoring which precedes a non-undoable refactoring without first undoing the non-undoable refactoring.

C.2 Emacs Tool

C.2.1 Requirements

The Emacs tool requires Emacs 20.x or 21.x and the Java Development Environment (JDE) version 2.2.9 or greater with the BeanShell. It also requires a configured and working gnuserv system with either gnuclient or gnudoit installed and in the path.

C.2.2 Installation

To use the tool the Lisp file `freefactor.el` must be loaded. This can be done when Emacs is started by inserting the following line into the `.emacs` file:

```lisp
(load-library "~/Project/emacs-tool/lisp/freefactor.el")
```
The following JAR files must also be in the class path for the BeanShell.

- freefactor.jar
- emacs-tool.jar
- transmogrify.jar

This can be achieved by adding the files to the project class path in the JDE configuration.

C.2.3 Configuration

The Emacs tool takes its configuration information from the JDE project file. Ensure that the prj.el file is configured correctly. Specifically the JDE must know the location of source and build paths.

C.2.4 Applying a Refactoring

Execute the command freefactor-refactor either by choosing the Refactor option from the Refactoring menu, or by typing

M-x freefactor-refactor

You will be prompted for the name of the refactoring to apply. The tool will then check the preconditions of the refactoring and apply it if they hold.

At any point during this process the tool may prompt for user input via the minibuffer or display failure information.

C.2.5 Testing a Refactoring

Execute the command freefactor-test either by choosing the Test option from the Refactoring menu, or by typing

M-x freefactor-test

You will be prompted for the name of the refactoring to test. The tool will then check the preconditions and display failures information as necessary.
Appendix D

Developer Guide

This chapter gives details on how to integrate the framework into a development tool.

D.1 Documentation

The Javadoc API documentation for the framework is available on the project web site (http://www.freefactor.org).

D.2 Tool Developers Guide

To use the framework, a tool must minimally contain a class which implements the Tool interface.

D.2.1 Implementation of the Tool Interface

- `addFailure(Condition condition)`
  This method is called by the framework when the Condition concerned is a failure for the refactoring being tested, i.e. it does not hold. It is intended that this should be added to a list of failures which can be displayed to the user in some way.

- `String getFileName(String packageName, String class)`
  This is used by the framework to obtain a valid filename for a new class. This is necessary because the framework does not store any information about the various directories where source files might be stored. It is intended that in most implementations this method will simply involve returning a path representing a subdirectory of the source directory corresponding to the package name.

- `String getSelectedPackage()`
  The tool must provide some interface for selecting packages for refactor-
ings which operate on packages. This method should return the selected package name as a String, or null if none is selected.

- 
  **Location getToolLocation()**
  This method should return a Location object representing the location of the cursor within the tool. The Location class is explained below.

- 
  **Location getSelectionStartLocation()**
  This method should return a Location object representing the location of the start of the selection within the tool, or null if no selection is active.

- 
  **Location getSelectionEndLocation()**
  This method should return a Location object representing the location of the end of the selection within the tool, or null if no selection is active.

- 
  **String getUserInput(String message)**
  This method is used by the framework to get input from the user. It is intended that this should be implemented with some kind of popup window in the tool giving the message and a text entry field.

- 
  **ConfirmationList displayConfirmationList(ConfirmationList l)**
  This method is used by the framework to display a confirmation list to the user. The tool should show the list as a table of keys and associated boolean values as checkboxes or similar constructions. The user should be allowed to modify the boolean values and the tool should return the modified ConfirmationList object (this does not necessarily have to be the same ConfirmationList object that was originally passed).

- 
  **void removeFileRequest(File file)**
  Some refactorings require files to be removed (for example when changing packages). This method is called by the framework to request a file to be removed. It is intended that the tool should store these files until the refactoring is complete. If the refactoring has completed without error, it should remove the files, possibly prompting for user confirmation.

### D.2.2 Use of the other tool classes

**RefactoringsFactory**

The RefactoringsFactory class can be used to obtain Refactoring objects to apply. The concrete subclass DefaultRefactoringsFactory makes all of the refactorings the framework knows about available. Tools can make their own subclasses which limit or augment the refactorings available. The factory returns RefactoringInfo objects to describe the refactorings available.

**ParserFactory**

The ParserFactory is used by the framework to get Parser objects. These are then used to get input streams for the files which need to be parsed. The
concrete subclass DefaultParserFactory returns DefaultParser objects which return input streams directly from the files themselves. A tool may subclass these classes in order to return input streams from different locations (for example the contents of a window in the tool when the file has not been saved). The parser factory is specified when the code model is created.

ChangeHistory
The ChangeHistory class represents a tree structure which can be used to represent the history of changes made to the code by the framework. The tree structure holds HistoryNode objects. It is envisaged that tools can use this structure as a model for a graphical display of the changes that have been made. The HistoryNode class can be subclassed to add more information if necessary.

PrintManager
The PrintManager interface provides a method for the framework to get a writer object for a particular file when printing the code model after refactoring. The tool should provide a class which implements this interface and returns writers for files as necessary. The print manager is specified when the code model is created.

D.2.3 The Refactoring Process
The refactoring process should be implemented in a similar way to the following:

- The first step is to create an instance of the CodeModel class. This encapsulates the framework’s representation of the code to be refactored. The tool should add all the files which need to be parsed to the model by calling the addFileToParse method.

- Use the RefactoringsFactory to list the refactorings available and display them to the user.

- When a user has selected a refactoring, the RefactoringsFactory should then be used to instantiate a Refactoring object.

- The parse method should then be called on the CodeModel instance to parse the code in preparation for refactoring. The parser factory specified will be used to obtain input streams for the files.

- The next stage is to obtain and check the precondition of the refactoring. To do so, the getPreCondition() method should be called to obtain a Condition object. The check method should then be called to see if the condition holds. If it does refactoring can continue, if not the process should stop and the failures should be displayed.

- Once the precondition has been checked, the Change objects for the refactoring should be obtained by calling the apply method. At this stage, RefactoringExceptions should be caught.
If the `Change` object has been obtained successfully, the changes should be applied to the `CodeModel` instance by calling the `change` method.

Once this has been done, the new code can be obtained by calling the `print` method on the `CodeModel` instance. The files will then be updated via the `PrintManager` instance which is in use.

**Undo**

The process for undo involves obtaining the `Change` object which corresponds to the undo change for the refactoring via the `getUndoChange` method. The process proceeds in the same way as the refactoring process above except that the postcondition should be checked instead of the precondition. The tool must also take care to only use the framework to undo undoable changes (determined by the `isUndoable` method. It must use its own mechanisms to undo changes which cannot be undone by the framework.

**D.3 Framework Developers Guide**

Refactorings within the framework are represented by three main components: the `Refactoring` class itself, the `Condition` classes and the `Change` classes.

Refactorings are created by creating a new concrete subclass of either the `PrimitiveRefactoring` class or the `CompositeRefactoring` class.

**D.3.1 Primitive Refactorings**

The main methods of interest are `getPreCondition`, `getPostCondition` and `apply`. Since the refactoring objects are stateful (i.e. there is one instance per refactoring applied), it is reasonable to collect information in the `getPreCondition` method to be used in the `apply` method.

The refactoring is given an instance of the `CodeModel` to use when it is instantiated. This can be used to obtain the various objects needed to perform the refactoring, for example the `CodeAnalyser`. During the refactoring process a `Tool` object is made available.

Another consideration is that the refactoring should make an instance of the `RefactoringInfo` class available via the `getRefactoringInfo` method. This should be held as a static member within the class.

**D.3.2 Composite Refactorings**

The `CompositeRefactoring` class has a member `refactorings` which contains the refactorings which make up the composite refactoring. This should be populated with the relevant objects when the refactoring is instantiated.

The `getRefactoringInfo` method should return a list of the composite refactorings represented by `RefactoringInfo` objects.
D.3.3 Conditions

Condition objects are used to check properties of the code for the pre and post conditions of a refactoring. There are many conditions already implemented in the framework, but it is likely that more may need to be created when implementing a new refactoring.

Creation of new conditions is a relatively simple process. The new class should be a subclass of the Condition class, and should implement the holds and getName methods minimally. The former returns true when the condition holds and false otherwise; the latter simply returns a name for the condition. The holds method is given an instance of the CodeModel and the Tool with which to check the condition.

D.3.4 Changes

In a similar way to conditions, there are many changes already implemented in the framework but more may need to be created.

To create a new change, a subclass of the Change class should be created. This should implement the apply method which applies the change to a given CodeModel. Arbitrary information may be supplied to the change when it is instantiated via the constructor.

The change should also implement the getUndoChange method which should return another Change object which corresponds to undoing the change. For example, the AddMethodChange returns a RemoveMethodChange as its undo change.
Appendix E

XML Refactoring Descriptions

The following sections give an example of how a refactoring can be described in XML using the framework, and the DTD (Document Type Definition) which the descriptions are validated against.

E.1 RenameMethod

The following XML file describes the RenameMethod refactoring in XML form.

```xml
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE refactorings SYSTEM "refactorings.dtd">
<refactorings>
   <refactoring name="RenameMethod" target="method">
      <precursor>
         <target valueid="methodDef"/>
         <userinput valueid="newMethodName" message="New method name"/>
         <pattern type="method" valueid="oldMethodPattern" def="methodDef"/>
         <pattern type="method" valueid="newMethodPattern" template="oldMethodPattern">  
            <name valueid="newMethodName"/>
         </pattern>
         <analysis name="MethodsToRename" valueid="methodsToRename">
            <param name="methodPattern" valueid="oldMethodPattern"/>
         </analysis>
      </precursor>
      <precondition>
         <condition name="NOT">
            <param name="condition">
               <condition name="MethodExists" expectedvalue="false">
                  <param name="methodPattern" valueid="newMethodPattern"/>
               </condition>
            </param>
         </condition>
      </precondition>
   </refactoring>
</refactorings>
```
E.2 Document Type Definition

The following file is a DTD for refactorings which can be described in the framework. It is used to validate files such as the one given previously.

```xml
<!-- DTD for refactoring descriptions -->
<!ELEMENT refactorings (refactoring)>  
<!ELEMENT refactoring (precursor, precondition, postcondition, changes)>  
<!ATTLIST refactoring
  name ID #REQUIRED
  target CDATA #REQUIRED>  
<!ELEMENT precursor (target?, userinput, pattern?, analysis)>  
<!ATTLIST precursor
  target? CDATA
  userinput EMPTY
  pattern? CDATA
  analysis CDATA>  
<!ELEMENT userinput EMPTY>  
<!ATTLIST userinput
  valueid ID #REQUIRED
  message CDATA #REQUIRED>  
<!ELEMENT pattern (target?, name?)>  
<!ATTLIST pattern
  target? CDATA
  name? CDATA
  expectedvalue CDATA
  methodPattern ID #REQUIRED
  valueid CDATA #REQUIRED>  
```
type (localvar\[classmember\]|method\[class\]|package) #REQUIRED
def IDREF #IMPLIED

<!ELEMENT target EMPTY>  
<!ATTLIST target valueid ID #REQUIRED>

<!ELEMENT name EMPTY>  
<!ATTLIST name valueid IDREF #REQUIRED>

<!ELEMENT analysis (param*)>  
<!ATTLIST analysis
  name CDATA #REQUIRED
  valueid ID #REQUIRED>

<!ELEMENT param (condition*, list*)>  
<!ATTLIST param
  name CDATA #REQUIRED
  valueid IDREF #IMPLIED>

<!ELEMENT condition (param*)>  
<!ATTLIST condition
  name CDATA #REQUIRED
  expectedvalue (true|false) #IMPLIED>

<!ELEMENT list ANY>

<!ELEMENT precondition (condition*)>

<!ELEMENT postcondition (condition*)>

<!ELEMENT changes (changes*, iterate*)>

<!ELEMENT change (param*)>  
<!ATTLIST change name CDATA #REQUIRED>

<!ELEMENT iterate (changes*)>  
<!ATTLIST iterate
  listid IDREF #REQUIRED
  valueid ID #REQUIRED>