Formalizing Refactorings
Implemented in Eclipse

Quinten David Soetens

Principal Adviser:  Prof. Dr. Serge Demeyer
Assistant Adviser:  Sylvain Degrandsart

Dissertation Submitted in May 2009 to the
Department of Mathematics and Computer Science
of the Faculty of Sciences, University of Antwerp,
in Partial Fulfillment of the Requirements
for the Degree of Master of Science.
## Contents

List of Figures ........................................ iv
List of Listings ........................................ vii
Nederlandstalige Samenvatting ......................... ix
Acknowledgements ....................................... xii
Abstract .............................................. xiii

1 Introduction ........................................ 1
  1.1 Definitions of Refactoring ......................... 2
  1.2 Reasons to Formalise Refactorings ................. 3
  1.3 Problem Statement ................................ 3
  1.4 Experimental Setup ................................ 4

2 Formal Specification Languages to Formalize Refactorings 6
  2.1 Introduction ..................................... 6
  2.2 The “Pull Up Method” Refactoring Informally .... 7
  2.3 Algebraic Specifications .......................... 8
  2.4 Graph Transformations ............................. 9
  2.5 Maude ............................................. 12
  2.6 Z .................................................. 14
  2.7 Comparison .................................... 16

3 Story Diagrams ...................................... 17
  3.1 Introduction .................................... 17
  3.2 Syntax and Semantics ............................ 17
3.2.1 Story Patterns ............................................. 17
3.2.2 Story Diagrams .......................................... 18
3.2.3 Additional story pattern constructs ..................... 18

4 The Eclipse Refactoring Framework 21
4.1 Eclipse Refactoring Architecture .......................... 21
   4.1.1 Normal Refactoring ................................... 22
   4.1.2 Processor Based Refactorings ......................... 23
   4.1.3 The Life Cycle of a Refactoring ...................... 23
4.2 Implementation of Pull Up Method ......................... 24
   4.2.1 Preconditions ....................................... 29
   4.2.2 Mechanics ........................................... 30
4.3 Implementation of Encapsulate Field ...................... 30
   4.3.1 Preconditions ....................................... 33
   4.3.2 Mechanics ........................................... 33
4.4 Implementation of Extract Class ......................... 34
   4.4.1 Preconditions ....................................... 37
   4.4.2 Mechanics ........................................... 38
4.5 Conclusions ............................................... 39

5 Formal Specification of Refactorings in Story Diagrams 40
5.1 Pull Up Method ........................................... 41
   5.1.1 Preconditions in Story Diagrams ....................... 41
   5.1.2 Mechanics in Story Diagrams ......................... 44
5.2 Encapsulate Field ......................................... 48
   5.2.1 Preconditions in Story Diagrams ....................... 48
   5.2.2 Mechanics in Story Diagrams ......................... 50
5.3 Extract Class ............................................... 54
   5.3.1 Preconditions in Story Diagrams ....................... 54
   5.3.2 Mechanics in Story Diagrams ......................... 56
5.4 Conclusions ............................................... 58

6 Validation of the Specifications 59
6.1 Testing Pull Up Method .................................... 60
   6.1.1 Main Scenarios ...................................... 60
   6.1.2 Further Testing ...................................... 69
6.2 Testing Encapsulate Field ................................ 71
   6.2.1 Main Scenario ...................................... 71
   6.2.2 Further Testing ...................................... 75
6.3 Testing Extract Class ..................................... 77
   6.3.1 Main Scenario ...................................... 77
   6.3.2 Further Testing ...................................... 82
List of Figures

1.1 Schematic view of the validation process. ...................... 5
2.1 Informal view of what the Pull Up Method refactoring does. . 7
2.2 Graph transformations P1 and P2 for the Pull Up Method refactoring. .......................................................... 11
2.3 Precondition 4 for the Pull Up Method refactoring. .......... 11
4.1 Communication Diagram showing the execution of a refactoring in the eclipse JDT, [25] .................................................. 24
4.2 1st dialog in the Pull Up refactoring wizard in eclipse. ...... 25
4.3 2nd dialog in the Pull Up refactoring wizard in eclipse. ...... 26
4.4 The dialog of the Encapsulate Field refactoring wizard in eclipse 31
4.5 The dialog of the Extract Class refactoring wizard in eclipse .... 35
5.1 Complete story diagram to check the preconditions of the Pull Up Method refactoring. ................................. 42
5.2 The story diagram that shows the mechanics of the Pull Up Method refactoring ................................. 45
5.3 The story diagram that makes target class abstract and creates a method stub in all its non abstract subclasses. .......... 46
5.4 The story diagram to check the preconditions of the Encapsulate Field refactoring. ................................. 49
5.5 The story diagram that shows the mechanics of the Encapsulate Field refactoring. ................................. 51
5.6 The story diagram to create a getter in the Encapsulate Field refactoring. .................................................. 52
B.11 The story diagram to create getters and setters for the Extract Class refactoring. ........................................ 106
B.12 The story diagram that changes field accesses to getter calls in the Extract Class refactoring. ............................ 107
B.13 The story diagram that changes field updates to setter calls in the Extract Class refactoring. ............................ 108
## List of Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Specification of <em>Pull Up Method</em> refactoring in Maude.</td>
<td>12</td>
</tr>
<tr>
<td>4.1</td>
<td>checkInitialConditions of the Pull Up refactoring</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>Extract of the createChangeManager code in the implementation of the Pull Up refactoring.</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>Extract of the code in the checkFinalConditions method of the Encapsulate Field refactoring.</td>
<td>32</td>
</tr>
<tr>
<td>4.4</td>
<td>Extract from the checkFinalConditions method in the Extract-ClassRefactoring that shows the precondition checks on the selected fields.</td>
<td>36</td>
</tr>
<tr>
<td>4.5</td>
<td>Extract from the performFieldRewrite method in the Extract-ClassRefactoring.</td>
<td>36</td>
</tr>
<tr>
<td>6.1</td>
<td>Java code for the Pull Up tests (conforming to the graph in figure 6.1).</td>
<td>63</td>
</tr>
<tr>
<td>6.2</td>
<td>Java code after pulling up method from Source and removing the similar method from Sibling2 and Sibling3 (corresponding to the graph in figure 6.2).</td>
<td>65</td>
</tr>
<tr>
<td>6.3</td>
<td>Java code after making the method abstract in GrandParent (corresponding to the graph in figure 6.3).</td>
<td>67</td>
</tr>
<tr>
<td>6.4</td>
<td>Extract of the source code that test the preconditions for the Pull Up Method refactoring.</td>
<td>69</td>
</tr>
<tr>
<td>6.5</td>
<td>Extract of the source code that test the mechanics for the Pull Up Method refactoring.</td>
<td>70</td>
</tr>
<tr>
<td>6.6</td>
<td>Java code for the Encapsulate Field tests (corresponding to the graph in figure 6.4).</td>
<td>72</td>
</tr>
<tr>
<td>6.7</td>
<td>Java code after Encapsulating i in class2. (corresponding to the graph in figure 6.5).</td>
<td>72</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.8</td>
<td>Extract of the JUnit tests to test the preconditions of the Encapsulate Field refactoring.</td>
<td>75</td>
</tr>
<tr>
<td>6.9</td>
<td>Extract of the JUnit tests to test the mechanics of the Encapsulate Field refactoring.</td>
<td>76</td>
</tr>
<tr>
<td>6.10</td>
<td>Java code for the Extract Class tests (corresponding to the graph in figure 6.6).</td>
<td>79</td>
</tr>
<tr>
<td>6.11</td>
<td>Java code after extracting field1 and field2 from class1 into a new nested Class. (corresponding to the graph in figure 6.7).</td>
<td>79</td>
</tr>
<tr>
<td>6.12</td>
<td>Extract of the JUnit tests to test the preconditions of the Extract Class refactoring.</td>
<td>82</td>
</tr>
<tr>
<td>6.13</td>
<td>Extract of the JUnit tests to test the mechanics of the Extract Class refactoring.</td>
<td>83</td>
</tr>
</tbody>
</table>
Refactoring is een belangrijk onderzoeksonderwerp binnen de software engineering. Refactoren is de programmacode aanpassen zodat de interne structuur verbeterd zonder het gedrag van het systeem te veranderen. Door te refactoren is de code van een systeem beter begrijpbaar en gemakkelijker aanpasbaar. Een refactoring is een vastgelegde methode om de code op een bepaalde manier aan te passen met een bepaald doel zonder dat het gedrag van het systeem veranderd. Dit doel kan zijn het verwijderen van duplicate code of het hernoemen van elementen om het gemakkelijker begrijpbaar te maken. Refactoren is dus het toepassen van een aantal refactorings.

Manueel refactoren is een taak die veel tijd in beslag neemt en waarbij gemakkelijk fouten kunnen gemaakt worden. Daarom zijn er tools ontwikkeld die refactorings automatisch kunnen uitvoeren. De populaire ontwikkelings omgeving, eclipse, biedt aan zijn gebruikers een uitgebreid gamma aan refactorings aan. Er is echter weinig documentatie over deze refactorings beschikbaar, laat staan een formele specificatie. Een formele specificatie van de refactorings die in eclipse geïmplementeerd zijn zou wel erg handig zijn.

Met een formele specificatie van refactorings zouden we wiskundig kunnen bewijzen dat refactorings bepaalde eigenschappen van de broncode behouden en andere eigenschappen veranderen. We zouden ook de pre- en postcondities kunnen bestuderen en zo de refactorings zelf analyseren.

Wat we dus in deze thesis proberen te achterhalen is of we de refactorings die geïmplementeerd zijn in eclipse kunnen reverse engineeren tot formele specificaties.

De precondities en postcondities van de refactorings kunnen vrij gemakkelijk...
lijk uit de broncode van eclipse worden afgeleid. Elke refactoring die geïmplementeerd is in eclipse moet een subklasse zijn van de abstracte klasse `Refactoring`. Deze declareert 3 abstracte methodes die in elke concrete refactoring moeten geïmplementeerd zijn, met name: `checkInitialConditions`, `checkFinalConditions` en `createChange`.

De `checkInitialConditions` en `checkFinalConditions` methodes, dienen beide om na te gaan of de precondities van de refactoring voldaan zijn. Als de precondities voldaan zijn, dan is de refactoring gegarandeerd gedragsbehoudend. De `createChange` methode dient dan om de veranderingen, die aan de code moeten aangebracht worden, samen te stellen. We kunnen dus voor elke refactoring eenvoudig de precondities vinden door in de broncode de methodes `checkInitialConditions` en `checkFinalConditions` te bestuderen. De postcondities vinden ligt net iets moeilijker omdat de veranderingen die moeten doorgevoerd worden reeds in de `checkFinalConditions` methode worden verzameld. Als hulp bij het ontdekken van de veranderingen kunnen we de refactoring uitvoeren en uittesten met variërende invoer. Op die manier kunnen onduidelijkheden uit de broncode worden verklaard.

Er zijn verschillende manieren die we kunnen gebruiken om refactorings te formaliseren. De meest belovende manier lijkt ons die van de graaf transformaties. Aangezien we programmacode kunnen voorstellen als een graaf, kunnen we een programma transformatie voorstellen als een graaf transformatie. Een refactoring is een speciaal soort programma transformatie en dus kunnen we refactorings voorstellen als graaf transformaties. In deze thesis hebben we ervoor gekozen om de refactorings voor te stellen als story diagrammen. Dit is een taal gebaseerd op graaf transformaties en heeft een goede tool support (Fujaba).

Doordat we Fujaba gebruiken om onze story diagrammen te maken, kunnen we code genereren uit de story diagrammen. Daardoor kunnen we de story diagrammen als uitvoerbaar beschouwen. We kunnen dus aantonen dat de story diagrammen die we hebben opgesteld een voorstelling zijn van de refactorings in eclipse. We doen dit door een klein stukje code te nemen en te vertalen naar een graaf. Nu kunnen we enerzijds op de code de refactoring in eclipse uitvoeren. En anderzijds kunnen we op de graaf de story diagramma voorstelling van de refactoring uitvoeren. Tenslotte moeten we nagaan of de verkregen graaf een graaf voorstelling is van de resulterende code.

Doordat Fujaba java code genereert voor onze story diagrammen, kunnen we de validatie automatiseren met JUnit\(^1\). De JUnit test gevallen voeren op de graaf voorstelling van een eenvoudig stukje code de story diagrammen uit.

\(^1\)JUnit is een framework om java code te testen.
en gebruiken daarna asserties om te testen dat de resulterende graaf is wat we verwachten. Wat we verwachten wordt bepaald door de refactoring uit te voeren in eclipse.

Om de precondities in de story diagrammen te testen gaan we een beetje anders te werk. Dan begint de JUnit test met het aanpassen van de graaf zodat één van de precondities niet voldaan is. De test gaat dan weer met asserties na dat het story diagram dat de precondities nagaat, faalt. Dit doen we voor elk van de precondities.

We kunnen concluderen dat het wel degelijk haalbaar is om de refactorings die geïmplementeerd zijn in eclipse te formaliseren.

Er zijn echter enkele problemen. Aangezien het meta-model dat we gebruiken voor de graaf voorstelling van java code niet volledig is, kunnen we niet alle java code als graaf voorstellen. Java syntax elementen die niet kunnen worden voorgesteld zijn ondermeer enumerations, annotations, generics, ...

Dan is er nog het probleem dat de story diagrammen een andere input methode gebruiken dan de refactorings in eclipse. In eclipse wordt de input voor de refactoring in twee fases ingegeven. Initieel wordt er in de programma-code de elementen geselecteerd waarop de refactoring zal werken, in het geval van de Encapsulate Field refactoring bijvoorbeeld wordt er een enkel attribuut van een klasse geselecteerd. Vervolgens kan er extra input ingegeven worden via de refactoring wizard. Bij de story diagrammen daarentegen wordt alle input tesamen ingegeven via parameters. Dit heeft tot gevolg dat er extra precondities nodig kunnen zijn die bij de implementatie van eclipse impliciet voldaan waren. Dit verschil in input methodes wordt pas echt een probleem als de refactoring op veel elementen moet werken. Vooral bij de Extract Method refactoring is dit duidelijk, hier wordt in eclipse een reeks expressies en statements uit de body van een methode geselecteerd, die geextraheerd worden in een nieuwe methode. Als we dit in story diagrammen zouden willen uitdrukken, dan moeten al die expressies en statements als een verzameling van parameters worden ingegeven en moet er in de story diagrammen een extra preconditie geplaatst worden, die na gaat dat alle expressies en statements die zijn meegegeven op elkaar volgen.

Tot slot willen we benadrukken dat story diagrammen als een visuele programmeertaal kan beschouwd worden. Dit houdt in dat alle problemen die er bij normaal programmeren opduiken ook hier van toepassing zijn. Een story diagram zal zelden van de eerste keer volledig correct zijn. Het moet getest worden en gedebugd in enkele iteraties. We kunnen echter nooit garanderen dat er in de story diagrammen geen bugs meer aanwezig zijn.
Acknowledgements

In order for every master student to successfully complete his studies he has to write a master thesis. The purpose is to show that the student is capable of performing a scientific research and report on his findings in a comprehensive document. This requires a lot of effort to complete successfully and I would therefor like to express my gratitude to a few people who helped and motivated me along the way.

First of all I would like to thank Prof. Dr. Serge Demeyer for giving me the opportunity to do my thesis on this subject.

I would also like to thank Sylvain Degrandsart for his valuable feedback and for helping me to steer this work in the right direction.

A special thank you goes out to all my colleagues and especially to David, with whom I had numerous conversations and important statistical comparisons like “Who used the most spaces in his text?” :-)

Finally I would also like thank my family and friends for their unconditional support and for always being there when I needed them and.

Thank you all.
The Eclipse IDE is a widely used programming environment. As a result the refactorings it provides for Java code are also very well known. However there is no formal definition available for Eclipse’s refactoring methods. Such a formal definition could be used to compare Eclipse’s implementation to others, and to prove certain properties on Eclipse’s refactoring methods. This leads us to the question of whether it is feasible to formalise the refactorings that are implemented in Eclipse. To answer this question the source code for a few refactorings is analysed to extract their pre- and postconditions. These pre- and postconditions are then written down in story diagrams which we found to be the most promising way of formalising refactorings. To validate that these story diagrams are equivalent to the implementation in Eclipse, both the story diagrams and the refactorings in Eclipse are executed, followed by a comparison of both outputs.
CHAPTER 1

Introduction

A development team of a software system is tasked with designing and implementing a product to meet the requirements of a client. When the product is finished the client puts the system to use. While the users work with the software, they note bugs that they encounter and they note features that they would like to see added to the system. These bug reports and feature requests are sent back to the development team which is now also charged with the maintenance of the system. The maintenance tasks include fixing bugs (corrective maintenance) and adding new features (perfective maintenance). When bugs are resolved and new functionalities are added to the system, a new version is released and put to use. All of this happens in iterations, and with each iteration adding new features to the system becomes increasingly harder. Indeed Lehman’s laws [17] teach us that a system has to change for it to remain useful and as it evolves it will become more and more complex. So the system that was neatly designed and implemented in the beginning has gradually become a “Big Ball Of Mud”[9]. One of the ways we can counteract this design decay, is by refactoring the code.

The eclipse IDE\(^1\) is a very popular programming environment that offers a wide range of automatic refactorings to aid the developers in their refactoring needs. However there is little documentation available on these refactorings, let alone a formal specification. A formal specification for the refactorings in eclipse could prove to be very usefull and we therefore ask ourselves if it is feasible to formalise the refactorings that are implemented in eclipse.

\(^1\)http://www.eclipse.org
1.1 Definitions of Refactoring

The term refactorings was coined by Opdyke and Johnson[21] as

Transformations of the source code that make it easier to understand and reuse, while preserving its behavior.

Opdyke later cataloged some low level refactorings for C++ code and he precisely described the preconditions and mechanics for each one.[20]

A more quoted definition for refactoring is the one by Martin Fowler[11], who actually gives two definitions for refactoring, one for the noun and one for the verb:

A refactoring (noun) is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.

Refactoring (verb) is the action to restructure software by applying a series of refactorings without changing its observable behavior.

In his book on refactoring[11] Fowler cataloged 72 refactorings and he mentions many more on the website he maintains[10].

Important to note is that refactorings do not change the observable behavior, however they could certainly change other aspects of the code. Indeed refactorings are even meant to optimize certain quality attributes of the code and/or the system. If they are performed well, refactoring improves the design of a system and makes it easier to understand and thus makes it easier to maintain and adapt. It also helps to find bugs in the system.

Manually performing refactorings is a time consuming and error-prone job[23]. So to further aid the developers, there exists a wide range of tools to automatically perform refactorings. These automated refactorings are made possible by the use of preconditions, that if satisfied guarantee that the system’s behavior is preserved[20].
1.2 Reasons to Formalise Refactorings

There are several reasons why we would want to formalize refactorings:

- Formalising different implementations for automated refactorings allows to easily compare the different implementations.

- With a formal model that describes the effects of a refactoring on the source code, we can also mathematically prove that refactorings preserve certain properties of the code and change other properties. Bart Du Bois for instance used a formal representation of refactorings to study the effect refactorings have on the coupling and the cohesion of the code[6][5].

- Furthermore we can use the formal model to analyse the pre- and post-conditions of the refactorings and thus study certain properties of the refactorings themselves.

- With a formal specification for refactoring, extracting the preconditions and postconditions of the refactoring is easier than extracting them directly from source code.

1.3 Problem Statement

Eclipse is a widely used and very popular programming environment. In fact it is considered to be the de facto standard Integrated Development Environment (IDE) for Java programs. It also provides a wide range of automated refactorings. However there is little documentation available on these refactorings. Thus it is often unclear to a programmer what the precise pre- and postconditions of a refactoring are.

This brings us to the question that we want to answer in this feasibility study.

Can the implementation of refactorings in eclipse be reverse engineered into a formal specification for java refactorings?
1.4 Experimental Setup

To answer this question we first start by looking at how the refactorings are implemented in eclipse. This comes down to reverse engineering the following eclipse plugins:

- `org.eclipse.jdt.ui`
- `org.eclipse.ltk.ui.refactoring`
- `org.eclipse.ltk.core.refactoring`

These plugins are open-source and freely available on a CVS repository\(^2\). We have checked out version v20080805 of the necessary plugins from this repository. In chapter 4 we explain how refactorings are implemented in eclipse.

To find the preconditions and the mechanics of a refactoring we analyse the source code and try the refactoring out on some sample code. Once we have the preconditions and the mechanics of a refactoring as it is implemented in eclipse, we formalise them using a formalisation technique. In chapter 2, we give an overview of a few different formalisation techniques we could use to formalise the refactorings. The next chapter (chapter 3) describes in detail the syntax and semantics of story diagrams which is the formalisation technique we chose to work with. In chapter 5 we give the story diagrams for a few refactorings.

All that is left is to validate that the story diagrams are equivalent to the implementation of the refactorings in eclipse. By using Fujaba\(^3\) to create the story diagrams we can generate code from them which allows us to actually execute the story diagrams. This way we can compare the output from the story diagrams to the output of performing the refactorings in eclipse. Given an equivalent input, these outputs should also be equivalent. Figure 1.1 shows a schematic view of the validation process. We take some sample code which we translate to a graph representation. We can then execute the story diagrams on this graph representation and execute the refactorings in eclipse on the sample code. The final step is to validate the output of the story diagrams (i.e. assert that the result of the story diagram is equivalent to the result of the refactoring in eclipse). We present the results of the validations for each of the selected refactorings in chapter 6.

---

\(^2\)http://wiki.eclipse.org/index.php/CVS_Howto
\(^3\)www.fujaba.de
1.4. EXPERIMENTAL SETUP

Figure 1.1: Schematic view of the validation process.

Case Selection

The refactorings that we chose to formalise are:

- **Pull Up Method**: Eliminate duplicate behavior in two or more subclasses. Move a method to a superclass of its declaring class or declare the method as abstract in the superclass.

- **Encapsulate Field**: Make a field private and provide getter and setter methods.

- **Extract Class**: Create a new class and move the relevant fields and methods from the old class into the new class.

These refactorings range from a moderately complex to a rather complex implementation in eclipse. Each of these refactorings also represents a different category of refactorings. The Pull Up Method refactoring is a refactoring that deals with generalization (i.e. manage the inheritance hierarchy of classes). The Encapsulate Field refactoring is a refactoring that organizes data in a class so that working with this data becomes easier. And the Extract Class refactoring is a refactoring that moves features between objects, this is a very powerful refactoring in redesigning the code (i.e. to correct design mistakes or to reassign responsibilities).
In this chapter we look at some different ways to formalise refactorings. Each section will cover one technique and will clarify that language using a running example: the Pull Up Method refactoring. Section 2.3 covers Du Bois’[5] way which uses Briand’s[1] meta-model to provide a formal specification in an algebraic way. Section 2.4 deals with Mens’ [19][18] formalism of describing refactorings as graph transformations. Section 2.5 explains how Garrido et al. [13] used Maude, a language based on rewriting logic, to specify refactorings[2]. In section 2.6 is explained how refactorings could be specified in the Z notation, which is a popular formal specification language based on set theory and predicate calculus. Finally the last section (sections 2.7) give a comparison of these formalizations and concludes this chapter.

2.1 Introduction

Van Lamsweerde[28] defines a formal specification as:

The expression, in some formal language and at some level of abstraction, of a collection of properties some system should satisfy.

A formal language is one which is made of three components: rules that determine the syntax of the language (i.e. the grammar), rules for interpreting
2.2. THE “PULL UP METHOD” REFACTORIZING INFORMALLY

In this chapter we use a simplified version of the Pull Up Method refactoring as a running example. This refactoring is a way to eliminate identical behavior in classes with a common superclass. In this refactoring a method is pulled up from a class to its superclass, declarations of similar methods in subclasses of the superclass are removed.

![Figure 2.1: Informal view of what the Pull Up Method refactoring does.](image)

In each of the following sections we will provide a formalisation of the following pre- and post conditions as an example. Given a method $m$, a class $source$ and a class $target$

**Preconditions**

1. $target$ and $source$ are different classes
2. $target$ is a direct superclass of $source$
3. $m$ is declared in $source$
4. $target$ doesn’t contain a method with the same signature as $m$
2.3. ALGEBRAIC SPECIFICATIONS

Postconditions

1. \( m \) is no longer declared in \texttt{source}

2. \( m \) is now declared in \texttt{target}, which is still a parent of \texttt{source}

3. All declarations of similar methods in sibling classes of \texttt{source} have also been removed.

The preconditions and postconditions are for the sake of the example a bit simplified. This simplification does not diminish the value of the example, as they just add more similar checks. Normally we would also include a check to see that the variables, types and methods accessed and called in \( m \) are reachable from \texttt{target}. And we would also include the postcondition that all calls to the methods, that were deleted, in subclasses of \texttt{target} are redirected to the method \( m \).

2.3 Algebraic Specifications

Briand[1] created a formal meta-model for an object oriented system in an algebraic way. This meta-model consists mostly of sets and defines all the elements in the system as a set. For instance an object oriented system consists of a set of classes \( C \). Suppose \( c \in C \) then we define the set \( \text{Parents}(c) \) as the set of direct superclasses of \( c \). In a similar fashion the sets \( \text{Children}(c) \), \( \text{Descendants}(c) \) and \( \text{Ancestors}(c) \) are defined.

Next the methods are defined as a number of sets. Firstly there is for each class \( c \in C \) a set of methods \( M(c) \). A method can come in many varieties, it can be virtual, public, private, inherited, overridden, newly added, ... All of these possibilities are defined as sets. For instance \( M_{\text{INH}}(c) \) is the set of methods \( c \) inherited from \( \text{Parents}(c) \), \( M_{\text{OVR}}(c) \) is the set of methods that \( c \) inherited from \( \text{Parent}(c) \), but which \( c \) has overridden and \( M_I(c) \) is the set of all the methods that are implemented in \( c \), i.e. the methods that \( c \) provides an implementation for.

In a similar way there exist set definitions for method invocations, attributes, attribute accesses and types. For a more precise and complete definition we refer to section 3.0 of Briand’s paper[1] or to Chapter 10 of Du Bois’ phd thesis[5].

Once the meta-model is defined the refactoring methods can be described in function of the meta-model. Du Bois[5] formalises three refactorings namely \textit{Extract Method}, \textit{Move Method} and \textit{Replace Method with Method Object}. 
2.4. GRAPH TRANSFORMATIONS

Example
Formally we specify the Pull Up Method refactoring as \textit{PullUpMethod}(c_s, m, c_t). With \( c_s \in C \) is the source class, i.e. the class where the method to be pulled up is declared, \( m \in M \) is the method from \( c_s \) that will be pulled up, and \( c_t \in C \) is the class whereto the method is moved. Given these parameters we can state that the precondition will look as follows:

**Preconditions**

1. \( c_s \neq c_t \)
2. \( c_t \in \text{Parents}(c_s) \)
3. \( m \in M_I(c_s) \)
4. \( \exists m_t \in M_I(c_t) \) such that \( \text{Signature}(m_t) = \text{Signature}(m) \)

To express the postcondition we need a way of comparing entities before and after the refactoring. This is done as in Z [24] where entities after the execution of the method are marked with an apostrophe. In our case the sets before a refactoring are referred to in the usual notation and the sets after a refactoring are marked with an apostrophe. For example the set of methods implemented in class \( c \) before the refactoring is referred to as \( M_I(c) \) and after the refactoring it is referred to as \( M'_I(c) \)

**Postconditions**

1. \( m \notin M'_I(c_s) \)
2. \( M'_I(c_t) = M_I(c_t) + \{m\} \)
3. \( \forall c \in \text{Children}(c_t) : \exists m_c \in M'_I(c_c) \) such that \( \text{Signature}(m_c) = \text{Signature}(m) \)

2.4 Graph Transformations
Tom Mens et al.[19; 18] suggested to use graph transformations to represent refactorings or more generally to represent source code transformations. Systems are represented as graphs and refactorings are represented as graph transformation rules. Such graph transformation rules consist of a left-hand side and a right-hand side. When such a transformation rule is applied to a graph representation of a system, the left-hand side specifies which parts of the graph
are selected for the transformation. The right-hand side specifies how these selected parts should become.

A graph can contain a number of different nodes and edges. Table 2.1 represents these different graph elements that can be used. For graph representations of real software systems this meta-model still needs to be extended with more language specific constructs and more detailed method bodies. Pérez et al. [22] has extended the meta-model into a meta-model for java program graphs, by adding constructs for visibility, interfaces, packages, ...

Table 2.1: Node type set $\Sigma = \{C, M, MD, V, VD, P, E\}$ and edge type set $\Delta = \{l, i, m, t, p, e, c, a, u\}$.

<table>
<thead>
<tr>
<th>Node Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Class</td>
</tr>
<tr>
<td>M</td>
<td>Method Signature</td>
</tr>
<tr>
<td>MD</td>
<td>Method Definition</td>
</tr>
<tr>
<td>V</td>
<td>Variable</td>
</tr>
<tr>
<td>VD</td>
<td>Variable Definition</td>
</tr>
<tr>
<td>P</td>
<td>Parameter of a method definition</td>
</tr>
<tr>
<td>E</td>
<td>Expression of a method definition</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Edge Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>l : M → MD</td>
<td>dynamic method lookup</td>
</tr>
<tr>
<td>V → VD</td>
<td>variable lookup</td>
</tr>
<tr>
<td>i : C → C</td>
<td>inheritance</td>
</tr>
<tr>
<td>m : VD → C</td>
<td>variable membership</td>
</tr>
<tr>
<td>MD → C</td>
<td>method membership</td>
</tr>
<tr>
<td>t : V → C</td>
<td>variable type</td>
</tr>
<tr>
<td>M → C</td>
<td>method return type</td>
</tr>
<tr>
<td>p : MD → P</td>
<td>parameter definition</td>
</tr>
<tr>
<td>P → C</td>
<td>parameter type</td>
</tr>
<tr>
<td>e : MD → E</td>
<td>expression in method definition</td>
</tr>
<tr>
<td>E → E</td>
<td>subexpression in method definition</td>
</tr>
<tr>
<td>c : E → M</td>
<td>(dynamic) method call</td>
</tr>
<tr>
<td>a : E → { V</td>
<td>P }</td>
</tr>
<tr>
<td>u : E → { V</td>
<td>P }</td>
</tr>
</tbody>
</table>

**Example**

The pull up method is expressed as two consecutive graph transformations $P1$ and $P2$ (figure 2.2). $P1$ states that the method named `name` is moved from some `Child` class to its `Parent` class. The second transformation shows that
each copy of this method in subclasses of \texttt{Parent} will be removed.

![Graph transformations P1 and P2 for the Pull Up Method refactoring.](image)

**Preconditions** The left-hand side of transformation \textit{P}1 already contains many of our preconditions. It states that The \texttt{Parent} and \texttt{Child} classes are different, that the \texttt{Child} class is a subclass of the \texttt{Parent} class and that the method to be pulled up is a member of the \texttt{Child} class. This takes care of preconditions 1, 2 and 3. However some preconditions need be expressed as illegal subgraphs. These are subgraphs that may not be present in order for the transformation to succeed. In figure 2.3 we see the specification for precondition 4 as an illegal subgraph. It states that the method called \texttt{name} should not yet occur in the \texttt{Parent} class.

![Precondition 4 for the Pull Up Method refactoring.](image)

**Postconditions** The postconditions are implicitly contained in the graph transformations. Instead of explicitly specifying the postconditions, these specify the mechanics of the refactoring.

Transformation \textit{P}1 handles postconditions 1 and 2. After performing this transformation the method will no longer be a member of the \texttt{Child} class, but be a member of the \texttt{Parent} class instead.

The last postcondition (postcondition 3) is handled in transformation \textit{P}2. This transformation states that all methods in subclasses of the \texttt{Parent} class that
have the same name as the method we pulled up, will be removed from their respective classes.

2.5 Maude

Maude was made to be a simple, highly expressive language that can achieve a performance competitive with other efficient programming languages. It is used for three different things[2]:

1. As a highly expressive and performant programming language.

2. To specify formal specifications for an algorithm, a system, a language or a formalism. As the formal specification is written in Maude, which is also a programming language, the specifications themselves are executable and can thus be used to simulate the system or as a preliminary prototype.

3. To analyse and verify different properties expressing formal requirements.

So it is more than just a formal specification language. However Garrido and Méseguer[13] use Maude as a formal language to specify and verify refactoring methods. They specify three refactorings Push Down Method, Pull Up Field and Rename Temporary.

Example

Here we will provide the specification for our running example Pull Up Method. Listing 2.1 shows the formal specification of this refactoring based on Garrido’s Push Down Method and Pull Up Field. Note that in order for this specification to be complete the java meta-model as well as all the used auxiliary operations (such as getMethod and subClasses) need also be specified in Maude.

Listing 2.1: Specification of Pull Up Method refactoring in Maude.

```plaintext
fmod PULL−UP−METHOD is
pr JAVA−REF . pr CLASS−REF−HELPERS .
var TS:Types . vars CN MN: Qid .
--- other variable declarations omitted
op PullUpMethod : Qid Qid Types −> JavaClassesRefactoring .
  eq Cl <= PullUpMethod(CN, MN, TS) = if precondsPullUpMethodHold(Cl, CN, MN, TS)
  then applyPullUpMethodHold(Cl, CN, MN, TS)
  else Cl fi .

op precondsPullUpMethodHold : Classes Qid Qid Types −> Bool .
```
This is a definition of a module named PULL-UP-METHOD. When some subclasses of a class CN define a method MN with parameter types TS this method will be moved to the class CN and removed from the subclasses. Line 6 defines our refactoring method. When applied to a set of classes Cl it will perform the transformation applyPullUpMethod only if the preconditions hold. If the preconditions check returned false it will return the same set of classes Cl.

Preconditions

1. The set of subclasses of CN is not empty.
   \[(\text{subclasses} (CN, Cl) =/= \text{noClass})\]

2. In this case, this is the same as the previous condition, since we do not specifically state which subclass of CN we will pull the method from.

3. At least one subclass of CN defines a method called MN.
   \[(\text{someClassesDefineMethod} (\text{subclasses} (CN, Cl), MN, TS)).\]

4. The class CN does not contain a method called MN.
   \[(\text{getMethod} ((\text{md Class} \ CN \ sp \ cb), MN, TS) == \text{noMember})\]

Postconditions

The postconditions are defined by lines 20 to 28. We do not explicitly check on the postconditions as they are implicitly specified by the specification of the refactoring’s mechanics. The method named MN with parameters TS will be moved from the set of subclasses of CN that define this method and moved to the singleton set containing the superclass.
2.6 Z

Z [24] is a well known formal specification language. It provides a way to specify software systems using at its core a typed first order set theory. As such it is similar to our algebraic specification yet more structured. If we want to use Z to specify program transformations or specifically to specify refactorings we also need to specify the meta-model for object oriented programs in Z. As this is not readily available we will provide a simplistic view on object oriented programming using classes and methods. This should be sufficient for our running example Pull Up Method.

The basic types are \[ \text{CLASS, METHOD} \]

The state space for the system should contain information about the structure of the system. The structure of the system consists of a set of classes and a set of methods. Additionally we provide relationships between these types. The inheritance relationships parents, children are defined between classes, a membership relationship should be defined between methods and classes and a call relationship exists between methods.

\[
\begin{align*}
\text{OOSystem} & \\
\text{classes} : \mathbb{P} \text{CLASS} & \\
\text{methods} : \mathbb{P} \text{METHOD} & \\
\text{parents} : \text{CLASS} \rightarrow \mathbb{P} \text{CLASS} & \\
\text{children} : \text{CLASS} \rightarrow \mathbb{P} \text{CLASS} & \\
\text{members} : \text{CLASS} \rightarrow \mathbb{P} \text{METHOD} & \\
\text{calls} : \text{METHOD} \rightarrow \mathbb{P} \text{METHOD} & \\
\end{align*}
\]

\[
\begin{align*}
\text{dom parents} &= \text{classes} \\
\text{ran parents} &\subset \mathbb{P} \text{classes} \\
\text{dom children} &= \text{classes} \\
\text{ran children} &\subset \mathbb{P} \text{classes} \\
\text{dom members} &= \text{classes} \\
\text{ran members} &\subset \mathbb{P} \text{methods} \\
\text{dom calls} &= \text{methods} \\
\text{ran calls} &\subset \mathbb{P} \text{methods} \\
\forall c \in \text{classes} : \forall p \in \text{parents}(c) : c \in \text{children}(p)
\end{align*}
\]

We basically state that an object oriented system consists of classes and methods that are related as follows. Each class has a set of parent- and a set of children classes within the system. Each class contains a set of methods as defined by the members relationship. And additionally each method calls a set of methods as defined by the calls relationship.
Example

We can now use this state space to specify our Pull Up Method refactoring as follows.

\[
\begin{align*}
\text{PullUpMethod} & \quad \delta \text{OOSystem} \\
\text{source}? & : \text{CLASS} \\
\text{target}? & : \text{CLASS} \\
\text{method}? & : \text{METHOD} \\
\text{source}? & \in \text{classes} \\
\text{target}? & \in \text{classes} \\
\text{method}? & \in \text{methods} \\
\text{source}? & \neq \text{target}? \\
\text{target}? & \in \text{parents}(\text{source}?) \\
\text{method}? & \in \text{members}(\text{source}?) \\
\text{method}? & \notin \text{members}(\text{target}?) \\
\text{members}' & = \text{members} \setminus \{(c, \text{method}?) : \text{CLASS} \times \text{METHOD} \mid c \in \text{children}(\text{target}?)\} \\
\text{members}' & = \text{members} \cup \{(\text{target}?, \text{method}?)\}
\end{align*}
\]

Preconditions  The preconditions here state that the source and the target classes as well as the method to be pulled up should be present in our system. Secondly we state that the source class and the target class are not the same class (precondition 1). Next we state that the source class is a subclass of the target class (precondition 2). And lastly we state that the method should be a member of the source class (precondition 3) and not a member of the target class (precondition 4).

Postconditions  The postconditions are stated in the last two lines. As with our algebraic specification entities after the execution of the operation are marked with an apostrophe. The postconditions here state that all subclasses of the target class should no longer contain the method (postconditions 1 and 3) and the method should now be a member of the target class (postcondition 2).
2.7 Comparison

The algebraic specification and Z are similar in that they both are based on relatively easy mathematical notions such as sets and functions. They are therefore easily understandable. Graph transformations are visually appealing, if one is familiar with the different node and edge types, a single glance at the graph transformation is often enough to understand what the transformation does. Maude however is a relatively unknown programming language, so both the person who has to specify the refactorings, as the people who want to read, understand and use the specifications need to learn this programming language. For Maude the big argument is that once the refactorings are specified, they are executable and they thus form a usable refactoring tool.[13]

What we can specify using the formalisation techniques is highly dependent on the meta-model. The algebraic meta-model proposed by Briand for instance does not support statements and expressions included in method bodies, nor does it support local variables. So such refactorings as Extract Variable or Inline Method would be difficult to express. With Z there was no meta-model readily available so this would have to be made from scratch. For Maude the meta-model would also have to be remade, since the meta-model defined by Garrido and Méseguer is not publicly available. For the graph transformations we can use the meta-model for java program graphs proposed by Javier Pérez [22], who extended Mens’ meta-model with java specific constructs.

Since program code can be represented in the forms of graphs it seems a good idea to represent refactorings (i.e. program code transformations) as graph transformations. This is a visual notation, which is usually more understandable than other notations. Additionally graph transformations are supported by many tools, e.g. AGG\(^1\), Fujaba\(^2\), MoTMoT\(^3\), VMTS\(^4\), ... Considering this we think that graph transformations are the most promising way to formalise refactorings. That is why we chose to use story diagrams to formalise refactorings, as they are based on graph transformations and they have a good tool support (Fujaba). In the next chapter we give an overview of the syntactical constructs that we use in our story diagrams in chapter 5.

---

\(^1\)http://user.cs.tu-berlin.de/~gragra/agg/
\(^2\)http://www.fujaba.de/
\(^3\)http://motmot.sourceforge.net/
\(^4\)http://www.vmts.aut.bme.hu/
3.1 Introduction

Story diagrams is a graph grammar language developed by Albert Zündorf and his research group at the university of Kassel[30][8]. It is a combination of activity diagrams and story patterns. It uses the notations of the activity diagrams to represent the control flow through the diagram. The story patterns are the specification of activities using special collaboration diagrams based on graph grammar theory [30][8]. This chapter will give an overview of the syntactical elements used in the story diagrams in chapter 5 and appendix B. A formal definition of all syntactical elements in story diagrams and their semantics can be found in Zündorf’s habilitation thesis: “Rigorous Object Oriented Software Development”[30].

3.2 Syntax and Semantics

3.2.1 Story Patterns

A story pattern can be seen as a complex boolean condition on bound and unbound variables. When a story pattern is executed, it tries to bind all its unbound variables and verifies if the links between the variables exist. If this is successful the specified modifications (if any) are performed and the story pat-
tern succeeds, otherwise it fails. These special modifications are graph rewrite rules (i.e. graph transformations). Generally speaking such a graph rewrite rule consists of a left-hand side and a right-hand side. The left-hand side describes the “before” situation i.e. the selection of the current object structure that is going to be modified. Whereas the right-hand side describes the “after” situation that shows how the selection of the current object structure should change. In story patterns however the left-hand side and the right-hand side of the graph rewrite rule are shown together in one picture. The performed changes are explicitly marked using $<\text{create}>$ and $<\text{destroy}>$ stereotypes. The left-hand side of the graph transformation can then be interpreted as the normal objects and edges together with the destroyed (in red) objects and edges. The right-hand side of the graph transformation similarly is interpreted as the normal objects and edges together with the created (in green) objects and edges.

### 3.2.2 Story Diagrams

Story diagrams as we mentioned earlier are a combination of activity diagrams and story patterns. This allows us to write complex operations by managing the control flow with branches and/or loops. The execution of a story diagram starts at the unique start activity, which is represented by a filled circle (●). It then proceeds by following the outgoing transition to the next activity. If an activity has multiple outgoing transitions they need to be guarded by mutual exclusive boolean expressions shown in square brackets (“[...]”). The special keywords success and failure can also be used as guards for transitions representing the successful or failed execution of a story pattern.

A diamond shaped activity (◇) expresses a branch. The transition from a branch guarded by the else keyword is the transition taken if all boolean expressions of the other transitions were evaluated to false.

When the execution reaches a stop activity (represented by a “bulls eye”, ○) the execution of the story diagram comes to a halt.

We use the story diagrams as a means to implement methods. As such they can have formal parameters and a return value. The formal parameters can be used in the story diagram as bound variables. And the return value (if the return type is not Void) needs to be expressed at the stop activity.

### 3.2.3 Additional story pattern constructs

**For-Each activities** are a special kind of story pattern. Generally a given graph may contain more than one subgraph that matches the left-hand side of the graph rewrite rule in a story pattern. A For-Each activity is a pattern that is executed for all possible bindings of the variables in the pattern. This means
that the pattern’s rewrite rules are applied to all of the subgraphs that match the left-hand side of the graph transformation in the pattern. The outgoing transitions of a For-Each activity can only be marked by the keywords each time and end. The end transition is taken when the pattern has been applied to all the possible variable bindings. The each time transition is taken for each of the variable bindings.

**Collaboration Statements**  A story pattern can contain UML collaboration messages to call methods. The order in which the methods are called is determined by the sequence number of the collaboration messages. Since we use story diagrams to specify methods, we can activate the execution of another story diagram. This allows us to make certain steps in the story diagram more abstract by extracting those steps to a different story diagram.

**Boolean Constraints**  In many cases we want to add some additional constraints to the selection of the subgraph upon which we want to perform the story pattern. Such additional constraints are expressed as boolean expressions in curly braces (“{...}”). A story pattern with boolean constraints is executed by searching for a subgraph that fits the basic story pattern and that additionally allows to evaluate all boolean constraints to true.

**Maybe clause**  In story patterns different objects in a graph rewrite rule must match different objects in the selection of the subgraph. However sometimes it is handy to allow that two different nodes in the pattern match to the same object. Graphically this is shown as a boolean constraint in which we place the keyword maybe followed by an equation over two or more nodes.

**Multi-Object Variables**  With multi object variables we can create edges for a set of nodes or delete edges attached to a set of nodes. We can also assign attribute values for a set of nodes or destroy the whole set of nodes. Graphically a multi object variable is represented as two stacked dashed boxes.

**Negative Graph Elements**  A negative graph element is used in the selection of the subgraph to represent elements that may not occur. A negative graph element is represented in the story patterns by crossing out the elements with a bold $\times$.

**Optional Graph Elements**  Optional graph elements are elements that may or may not occur. In the story patterns they are represented using dashed shapes or lines. Optional variables are represented as dashed boxes. Optional edges are represented as a dashed line.
Path Expressions  A simple path expression is a dotted list of edge labels. It is evaluated by traversing the corresponding links. For instance if we have three nodes A, B and C and two edges labeled ab (between A and B) and bc (between B and C). Then we can traverse from A to C by the path expression “ab.bc”. The * operator can be used to compute the transitive closure of a basic path expression. So if we add another node D that is connected to C by the label bc. Then the path expression “ab.(bc*)” leads us from A to both C and D. So using a path expression in combination with a “For-Each” activity we can perform a story pattern for all nodes that are reachable by the path expression. Graphically this is represented by connecting two nodes with a double sided arrow (⇒) labeled with the path expression.

Multilinks  Multilinks are a way of dealing with ordered associations. This means that there is an order between certain edges. So multilinks introduce special operators to look up the next or indirect next objects in an ordered association. Graphically this is represented by a link from one edge to the other labeled by the keyword next and a small filled triangle (▷) indicating the direction of the order.
In this chapter we give a general overview of how refactorings are implemented in eclipse. For a more thorough description, we refer to the master thesis (chapter 2 and appendix A) of Jeroen van den Bos [25], who gives a reconstruction of the eclipse JDT refactoring architecture.

Furthermore we list in this chapter the preconditions and the mechanics we found for each of the refactorings that we selected. These are found by analysing the source code for the respective refactorings and trying the refactorings out on some sample code.

#### 4.1 Eclipse Refactoring Architecture

In eclipse the refactorings benefit from a refactoring framework supplied by the language independent refactoring Language Toolkit (LTK). This is provided by the plug-in org.eclipse.ltk.core.refactoring and its user-interface counterpart org.eclipse.ltk.ui.refactoring. The eclipse Java Development Tools (JDT) offers a rich variety of automated refactorings by extending the ltk in the plug-in org.eclipse.jdt.ui.[25][29][12]

In the ltk most of the functionality is implemented in abstract classes, that follow the “Template” design pattern. The refactoring code in the jdt is therefore meant to fill in the gaps with refactoring specific functionality.[29]
4.1. Normal Refactoring

Each concrete refactoring is required to extend the abstract class
\texttt{org.eclipse.ltk.core.refactoring.Refactoring}. As such each refactoring
needs to implement the three template methods provided by the
\texttt{Refactoring} superclass:

1. \texttt{checkInitialConditions(IProgressMonitor)} : \texttt{RefactoringStatus}
2. \texttt{checkFinalConditions(IProgressMonitor)} : \texttt{RefactoringStatus}
3. \texttt{createChange(IProgressMonitor)} : \texttt{Change}

The first one (\texttt{checkInitialConditions}) is called when the refactoring is
launched and is used to check some basic preconditions. Typically this checks
that the workspace is in a consistent state (i.e. that there are no compilation
errors). This method should be short-running since the result may determine
the way the user interface of the refactoring behaves on startup. It returns
a \texttt{RefactoringStatus} object. This is used to communicate the result of the
precondition checks to the refactoring execution entities. If the returned status
has the severity of \texttt{RefactoringStatus\#FATAL} the refactoring will be consid-
ered as not being executable.

After \texttt{checkInitialConditions} was successfully performed and we have
entered all necessary input in the refactoring wizard. The \texttt{checkFinalCondi-
tions} method is called to check the remaining preconditions. In a Rename
refactoring for instance the new name of the element was provided as ad-
ditional input. The \texttt{checkFinalConditions} will then need to verify that
no element with that name already exists. Additionally to the precondition
checking this method also collects the necessary data for the change genera-
tion. This is because the final precondition checking almost always performs
the same computations as the change generation. So during the precondition
checking the changes that will be performed on the workspace are gradu-
ally constructed and stored as a set of change descriptions. Later on in
the change generation these change descriptions need merely be retrieved and
merged into a single \texttt{Change} object. The \texttt{checkFinalConditions} method
also returns a \texttt{RefactoringStatus} object. If the status had a severity of
\texttt{RefactoringStatus\#FATAL} the precondition checking is terminated and the
unsatisfied preconditions are communicated back to the user.

Finally if all preconditions are satisfied, the \texttt{createChange} method is called.
This returns a change object (an object of type \texttt{org.eclipse.ltk.core.refac-
toring.Change}), based on the change descriptions created in the \texttt{checkFinal-
Conditions} method. The change object can then be used by the refactoring
user-interface to present us with a preview of the changes. The change object is also used by the core refactoring framework to actually apply the needed changes on the workspace.

4.1.2 Processor Based Refactorings

Some refactorings have an impact on files or elements in other parts of the eclipse environment. For instance if a breakpoint is placed on a line of code which is moved to a different source file by a refactoring, then the breakpoint should be moved as well. The implementation of the refactoring however shouldn’t deal with each of these concerns. To facilitate this the ltk provides so called processor based refactorings. These are basically the same as the normal refactorings except that with each call to a method in their interface they call a list of participants in order to keep them up to date. The actual refactoring itself extends the abstract class `org.eclipse.ltk.core.refactoring.participants.RefactoringProcessor`, which provides the same three template methods as the `Refactoring` base class.

4.1.3 The Life Cycle of a Refactoring

To summarise we provide an overview of the predefined procedure that each refactoring follows. Figure 4.1 is a UML collaboration diagram that shows how a refactoring is executed.

With each selection a user makes in the editor or any of the source code views (e.g. the outline, the package explorer, ...) in the eclipse environment the refactoring’s Action object is allowed to enable or disable itself based on the current selection (1: `selectionChanged()`). For instance when selecting a class, the Extract Method refactoring will be disabled since it cannot be performed on a class. This determines which refactorings will appear in the refactoring submenu of the popup menu when right clicking. When the user wants to execute an available refactoring he chooses the refactoring either from the dropdown refactor menu, or from the refactoring submenu in the context menu. At this point the run method on the refactoring’s Action object is executed (2: `run()`) which delegates the execution to the refactoring starter object (2.1: `run()`). The refactoring starter then instantiates the actual refactoring object and calls the `checkInitialConditions` method (2.1.1: `checkInitialConditions()`). If the initial preconditions were satisfied the refactoring wizard is opened (2.1.2: `open()`) to allow the user to enter additional input for the refactoring. When the wizard has all necessary information the remaining preconditions are checked (2.1.2.1: `checkFinalConditions()`) and...
if they are satisfied the change object is created (2.1.2.2: `createChange()`) and 2.1.2.2.1: `create()`). This change object encapsulates all the changes that will be performed on the workspace. Using this change object the wizard is capable of providing the user with a preview of the source code changes that will be applied. If the user approves, the wizard can then use the same change object to perform the changes to the actual workspace (2.1.2.3: `perform()`).

![Communication Diagram showing the execution of a refactoring in the eclipse JDT][25]

**Figure 4.1:** Communication Diagram showing the execution of a refactoring in the eclipse JDT.[25]

### 4.2 Implementation of Pull Up Method

The Pull Up refactoring is implemented in the class

```
PullUpRefactoringProcessor
```

Which can be found in the following package of the eclipse jdt.

```
org.eclipse.jdt.internal.corext.refactoring.structure
```

The Pull Up refactoring in eclipse allows to pull up several elements from a class at the same time and elements of different kinds (Methods, Fields and Types) can be pulled up together. We, however, are only interested in how the Pull Up works for Pulling Up a single Method. To pull up a method in
eclipse, we select the method in any of the source code views and then we select the Pull Up refactoring from the refactoring menu. The Pull Up refactoring wizard will then provide us with a dialog (see figure 4.2) in which we can choose to which superclass we wants to pull up the selected method. Instead of actually moving the method from the declaring type to a superclass, we are also presented with the option of declaring the method abstract in the destination type. This will make the destination class abstract and optionally create method stubs of the abstract method in the non-abstract subclasses of the destination class.

![Figure 4.2: 1st dialog in the Pull Up refactoring wizard in eclipse.](image)

If we chose “pull up” as action for our method we are presented with the next dialog (see figure 4.3). In this dialog we can choose which of the methods in subclasses of the targeted superclass we wish to remove. We can only remove methods that have the same signature as the method that will be pulled up.

In what follows we will refer to the method to be pulled up as Method, the class from which it will be pulled (i.e. the declaring class) as Source and
4.2. IMPLEMENTATION OF PULL UP METHOD

Figure 4.3: 2nd dialog in the Pull Up refactoring wizard in eclipse.

the class to which it is pulled (i.e. the destination class) as Target. The set of methods that are marked for deletion in the second dialog is referred to as deleteMethods. We will refer to the set of methods in subclasses of Target, that have the same signature as Method and are not in the set deleteMethods, as keepMethods.

In listing 4.1 we show the implementation of the checkInitialConditions method. At this point in the refactoring’s life cycle we only have Method and Source as input. We see on line 895 and line 898, that it delegates its checks to two other methods: protected RefactoringStatus checkDeclaringType(final IProgressMonitor monitor) and protected RefactoringStatus checkIfMembersExist(). This first one checks whether Source is not any of the following things: an enumeration, an interface or an annotation. It also checks that Source is not binary or read-only and that it is not the class java.lang.Object. Additionally it checks that Source has at least one super-class that is not binary and is not the class java.lang.Object. The second
method (checkIfMembersExist()) checks if Method, does in fact exist in the internal model of the source code.

Listing 4.1: checkInitialConditions of the Pull Up refactoring

```java
891 public RefactoringStatus checkInitialConditions(final IProgressMonitor monitor) throws CoreException, OperationCanceledException {
892     try {
893         monitor.beginTask(RefactoringCoreMessages.PullUpRefactoring_checking, 1);
894         final RefactoringStatus status = new RefactoringStatus();
895         status.merge(checkDeclaringType(new SubProgressMonitor(monitor, 1)));
896         if (status.hasFatalError())
897             return status;
898         status.merge(checkIfMembersExist());
899         if (status.hasFatalError())
900             return status;
901         return status;
902     } finally {
903         monitor.done();
904     }
905 }
```

If the preliminary preconditions were met we are presented with the refactoring wizard’s dialogs (figures 4.2 and 4.3) to provide additional input (i.e. Target and deleteMethods). The remaining preconditions are then checked by the checkFinalConditions method which also works by delegating the precondition checks to several methods:

The methods checkGenericDeclaringType() and checkFinalFields() do not interest us, since our meta-model (see appendix A) does not support generic types and we only want to pull up a Method and not Fields. The method checkAccesses() checks that all fields, methods and types accessed in Method are accessible from Target. The method checkMembersInTypeAndAllSubtypes() firstly checks that a method similar to (i.e. a method with the same signature) Method is not already declared in the destination type. Secondly it checks the return types and visibility of the methods in keepMethods. The return type of such a method should equal the return type of Method. The visibility of such a method may not be private and if the class that declared that method is in a different package than Target, then the visibility may also not be default. The method checkIfSkippingOverElements() checks that the method is not declared in any of the skipped classes (a skipped class is a class that is both a superclass of Source and a subclass of Target). Finally if we will make Target abstract by either selecting the action “declare abstract in destination” or by pulling up an abstract method and Target is not already abstract the method checkConstructorCalls() is called. This is to verify that Target is not instantiated anywhere. Because if a class is instantiated, we can’t make it abstract.
The changes are created at the end of the `checkFinalConditions()` method by calling the `createChangeManager()` method. This method creates the changes that will be made and puts them in a `TextEditBasedChangeManager` object. This change manager is then later used in the `createChange()` method to collect all changes and merge them into a single `Change` object.

In listing 4.2 we show an extract of the `createChangeManager()` method. What happens here is we iterate over all compilation units that are affected by the refactoring. If the unit is the target (i.e. the destination class) then we iterate the members that will be moved. If one of those members is a method, then a copy of that method is created (i.e. `newMethod` is a copy of `oldMethod`). Then we say to the `ASTRewrite` object that is linked to the target’s `CompilationUnitRewrite` object that this new method will need to be inserted in the AST\(^1\) of the target’s compilation unit. Internally all changes to each compilation unit are kept in a `CompilationUnitRewrite` object that is mapped to their respective compilation unit in the `fCompilationUnitRewrites` map. At the end of the `createChangeManager()` method the change manager is created and all changes (i.e. all compilation unit rewrites) are registered in this manager.

The other changes that are made by the Pull Up refactoring can also be found by tracing the statements an the method calls in the `createChangeManager()` method. For instance if the affected compilation unit (`unit`) was one that contained one of the methods that needed to be removed, then the method `deleteDeclarationNodes()` is called. This method says to the `ASTRewrite` object that this method needs to be removed (`rewriter.remove(node,...);`). Some other methods that aid in the creation of the rewrites (i.e. changes) are `createAbstractMethod()` and `addNecessaryMethodStubs()`.

Listing 4.2: Extract of the createChangeManager code in the implementation of the Pull Up refactoring.

```java
1073     final ICompilationUnit target = destination.getCompilationUnit();
1078     final ICompilationUnit [] units = getAffectedCompilationUnits(...);
1085     for (int index = 0; index < units.length; index++) {
1086         ICompilationUnit unit = units[index];
1091         CompilationUnitRewrite rewrite = getCompilationUnitRewrite(...)
1097         fCompilationUnitRewrites, unit);
1104         if (unit.equals(target)) {
1105             final ASTRewrite rewriter = rewrite.getASTRewrite();
1116             for (int offset = fMembersToMove.length - 1; offset >= 0; offset--)
1117             member = fMembersToMove[offset];
1120             } else if (member instanceof IMethod) {
1140             final MethodDeclaration oldMethod = ASTNodeSearchUtil.get
1141             MethodDeclarationNode((IMethod) member,...));
1142             if (oldMethod != null) {
```

\(^1\)The Abstract Syntax Tree (AST) is a tree representation of the abstract (simplified) syntactic structure of the source code
4.2. IMPLEMENTATION OF PULL UP METHOD

```java
final MethodDeclaration newMethod = createNewMethodDeclarationNode(..., oldMethod, ...);
rewriter.getListRewrite(declaration, ...).insertAt(newMethod, ...) =
ImportRewriteUtil.addImports(rewrite, oldMethod, ...);
```

4.2.1 Preconditions

The preconditions for the Pull Up Method refactoring that we found are:

1. `Source` is not an enumeration.
2. `Source` is not an annotation.
3. `Source` is not an interface.
4. `Source` is not binary.
5. `Source` is not read-only.
6. `Source` is not the class `java.lang.Object`.
7. `Source` has at least one non binary superclass that is not the class `java.lang.Object`.
8. `Method` exists and is not null.
9. `Method` is not a constructor for `Source`.
10. All fields, methods and types accessed in `Method` are accessible from `Target`.
11. `Target` does not already declare a method with the same signature as `Method`.
12. Methods in the set `keepMethods` have the same return type as `Method`.
13. Methods in the set `keepMethods` are not declared private.
14. Methods in the set `keepMethods`, that are declared in a class that belongs to a different package than `Target`, should not have visibility default.
15. A method with the same signature as `Method` may not be declared in skipped classes.
16. If the action “declare abstract in destination” is selected or Method is abstract, then Target may not be instantiated.

4.2.2 Mechanics
The mechanics for the Pull Up Method refactoring that we identified are:

1. If the action “declare abstract in destination” is selected.
   1.1 Method remains in Source.
   1.2 Target is made abstract and an abstract declaration of Method is added to Target.
   1.3 The necessary stub methods are created in the non-abstract subclasses of target.

2. If the action “pull up” is selected.
   2.1 Method is moved from Source to Target
   2.2 If Method was abstract, Target will be made abstract and the necessary method stubs will be added to the non-abstract subclasses of Target.
   2.3 The methods in the set deleteMethods are removed from their respective classes.

4.3 Implementation of Encapsulate Field
The Encapsulate Field refactoring is implemented in the class

```
SelfEncapsulateFieldRefactoring
```

Which can be found in the following package of the eclipse jdt.

```
org.eclipse.jdt.internal.corext.refactoring.sef
```

This refactoring is meant to make an attribute of a class private and generate accessor methods for it. To apply this refactoring, we select the field we wish to hide in one of the source code views or the editor and then we start the refactoring by selecting it from the refactoring menu. We are then presented with the Encapsulate Field refactoring wizard dialog as shown in figure 4.4. Here we need to enter the names we wish to use for our accessor methods. And we also need to specify whether we want to use the accessors in the declaring
4.3. IMPLEMENTATION OF ENCAPSULATE FIELD

In what follows we will refer to the field that will be encapsulated as Field and the names that are entered for the setter and the getter as respectively SetterName and GetterName.

In this refactoring the checkInitialConditions method is merely used to verify that there are no compilation errors and that Field exists and that it has a valid type.

The more interesting preconditions are checked in the method checkFinalConditions(). Firstly it looks for any methods with the name SetterName or GetterName. If such a method is found, then that will be used as the accessor. Once this has been established it calls the checkMethodNames() method, which verifies for both SetterName and GetterName that they are not an empty string and that they are a valid java identifier\(^2\). Additionally it also checks that if an existing getter or setter method is used, that this has the same static property as Field. Further down, the method checkInHierarchy() is called this checks that if a new method will be created that it does not override

---

\(^2\) In Java, all identifiers must begin with a letter, an underscore, or a Unicode currency character. Any other symbol, such as a number, is not valid. Furthermore, an identifier cannot have the same spelling as one of Java’s reserved words.
any existing method. It also checks the return types of the existing getter/setter methods. In the case of a getter this should be the same type as Field and in the case of a setter this should be void.

The mechanics are also prepared in the checkFinalConditions() method. This way the createChange() method merely needs to collect all changes from the change manager and merge them into a single Change object. In listing 4.3 we show an extract of the code in the checkFinalConditions() method where the changes are created. We iterate all affected compilation units and for each such unit we send an AccessAnalyser object to analyse the AST, representing the compilation unit, using the visitor design pattern\(^3\). This AccessAnalyser object searches for references to Field and for each such reference it determines how to change this reference into a getter or setter call.

At the end of the iteration if the current compilation unit was not the one representing the declaring class, then the changes can be created and added to the change manager (this is done in the createEdits() method). If however the compilation unit was the class that declared Field then some more changes need to be added. These changes are calculated in the addGetterSetterChanges() method, where a getter method and setter method are created and added to the declaring class. Additionally this method also changes the visibility of Field to private.

Listing 4.3: Extract of the code in the checkFinalConditions method of the Encapsulate Field refactoring.

```
359 ICompilationUnit owner = fField.getCompilationUnit();
362 for (int i = 0; i < affectedCUs.length; i++) {
363   ICompilationUnit unit = affectedCUs[i];
364   CompilationUnit root = null;
365   List descriptions = null;
366   if (owner.equals(unit)) {
367     root = fRoot;
368     descriptions = ownerDescriptions;
369   } else {
370     root = new RefactoringASTParser(...).parse(unit, true);
371     descriptions = new ArrayList();
372   }
373   AccessAnalyzer analyzer = new AccessAnalyzer(...);
376   descriptions.addAll(analyzer.getGroupDescriptions());
378   if (!owner.equals(unit))
380     createEdits(unit, ..., descriptions, ...);
387   ownerDescriptions.addAll(addGetterSetterChanges(..., usingLocalSetter, →
388     usingLocalGetter));
395   createEdits(owner, ..., ownerDescriptions, ...);  
```

\(^3\)The visitor design pattern is a way of separating an algorithm (e.g. the access analyser) from an object structure (e.g. the AST) upon which it operates.
4.3. IMPLEMENTATION OF ENCAPSULATE FIELD

4.3.1 Preconditions

The preconditions that we found for the Encapsulate Field refactoring are:

1. *Field* has a valid type.

2. *GetterName* and *SetterName* are not the empty string.

3. *GetterName* and *SetterName* are valid Java identifiers.

4. An existing getter/setter has the same static property as *Field*.

5. An existing getter should have as type the type of *Field*.

6. An existing setter should have as type void.

7. If a new getter/setter will be created, this may not override an existing getter/setter of a superclass.

4.3.2 Mechanics

The source code changes that we found for the Encapsulate Field refactoring are:

1. *Field* will be made private.

2. If necessary a new setter/getter will be created with the same static property of *Field*. The visibility of the new getter/setter is dependent on the original visibility of *Field*.
   - if *Field* was public, the new getter/setter will be made public.
   - if *Field* was protected, the user can choose public or protected in the refactoring dialog.
   - if *Field* was default, the user can choose public or default in the refactoring dialog.
   - if *Field* was private, the user can choose any visibility in the refactoring dialog.

3. All accesses of *Field* outside the declaring class, will be changed into a call to the getter method.

4. All updates of *Field* outside the declaring class, will be changed into a call to the setter method.

5. If the user chose the option “use setter and getter” as the way to access *Field* in the declaring type then all accesses and updates of *Field* inside the declaring class, will also be changed into calls to the getter and setter methods. With of course the exception of the accesses and updates inside the getter and setter methods themselves.
4.4 Implementation of Extract Class

The Extract Class refactoring is implemented in the class

    ExtractClassRefactoring

Which can be found in the following package of the eclipse jdt.

    org.eclipse.jdt.internal.corext.refactoring.structure

This refactoring creates a new class and moves a number of selected fields
to this new class with the added option of creating getters and setters for these
fields. Accesses to these fields are now redirected through a new field that has
the new class as type. As such this refactoring can be viewed as a glorified En-
capsulate Field refactoring. However in combination with the “Move Method”
to move behavior closer to its data, this refactoring is a powerful tool to split
a complicated class into two classes with clearer responsibilities.

To perform this refactoring in eclipse we select a class in the editor or any
of the source code views and select Extract Class from the refactoring menu.
If the class has at least one usable field the extract class refactoring wizard
presents us with the refactoring dialog (as seen in figure 4.4). In this dialog we
need to enter a name for the new class and a name for the new attribute that
has the new class as type. We can also choose to either make the new class a
nested class inside the original class or create it in the package that contains
the original class. We also need to select at least one field that we wish to
extract into the new class.

In what follows we will refer to the selected class as $\text{Source}$, the set of se-
lected fields as $\text{Fields}$, the new class as $\text{NewClass}$, the new attribute in class
$\text{Source}$ of type $\text{NewClass}$ as $\text{NewField}$.

Initially the refactoring only has $\text{Source}$ as input. The checkInitalCondi-
tions() method again checks some preliminary preconditions on the initial in-
put. It checks that $\text{Source}$ exists and that it is modifiable. It also checks that
$\text{Source}$ has at least one field that can be moved to $\text{NewClass}$.

After having entered everything in the dialog the refactoring has all neces-
sary input and can now check the rest of the preconditions. The checkFinal-
Conditions() starts by calling the validateAll() method of the nested class
ExtractClassDescriptorVerification. This method in turn calls the meth-
ods validateClassName(), validateFields() and validateParameterName().
4.4. IMPLEMENTATION OF EXTRACT CLASS

This first one (validateClassName()) checks that the name we chose for NewClass is a valid java type name and it checks that a class like NewClass does not yet exist. This means that if we chose to make NewClass a nested class, then Class may not already contain a method with the name of NewClass and if we chose to make NewClass top level then a class with the same name as NewClass should not already exist in the package that contains Class.

The second method (validateFields()) checks that the set Fields contains at least one field, and that no two fields in the set Fields will get the same name in NewClass. This last one needs to be checked since it is possible to change the field names by pressing the edit button in the dialog in figure 4.5.

The last method (validateParameterName()) is to check that the name we entered for NewField is a valid java identifier and that no field with that name already exists in Class.

Next the checkFinalConditions() method verifies that the fields in set Fields can be moved. The code snippet in listing 4.4 shows these checks. The
fields that will be moved may not be static, transient or volatile.

Listing 4.4: Extract from the checkFinalConditions method in the ExtractClassRefactoring that shows the precondition checks on the selected fields.

```java
for (Iterator iter = fVariables.values().iterator(); iter.hasNext();) {
    FieldInfo fi = (FieldInfo) iter.next();
    boolean createField = isCreateField(fi);
    if (createField) {
        if (Flags.isStatic(flags)) {
            status.addFatalError(ExtractClassRefactoring_error_field_is_static);
            return result;
        }
        if (Flags.isTransient(flags)) {
            result.addWarning(ExtractClassRefactoring_warning_field_is_transient);
        }
        if (Flags.isVolatile(flags)) {
            result.addWarning(ExtractClassRefactoring_warning_field_is_volatile);
        }
    }
}
```

Furthermore the checkFinalConditions() method calls performFieldRewrite() , which by its name doesn’t look like a precondition check, but it does check some more things along the way as is shown in listing 4.5. It verifies that, if NewClass will be created top level, the type of each field in Fields is not private and if the type of such a field is protected, it should not be declared in a class that is in a different package than where NewClass will be created. This is to make sure that the types of the fields are visible from NewClass. For instance if one of the fields has as type a private nested class in Source then that type is not visible from a top level class. It would however be visible from a nested class in Source, which is why this check is only done when NewClass will be made top level.

Listing 4.5: Extract from the performFieldRewrite method in the ExtractClassRefactoring.

```java
if (fDescriptor.isCreateTopLevel()) {
    IVariableBinding binding = vdf.resolveBinding();
    ITypeRoot typeRoot = fBaseCURewrite.getCu();
    if ((binding == null || binding.getType() == null) {
        status.addFatalError(ExtractClassRefactoring_fatal_error_cannot_resolve_binding);
    } else {
        ITypeBinding typeBinding = binding.getType();
        if (Modifier.isPrivate(typeBinding.getDeclaredModifiers())){
            status.addError(ExtractClassRefactoring_error_referencing_private_class);
        } else if (Modifier.isProtected(typeBinding.getDeclaredModifiers())){
```
In a similar way to the previous two refactorings, the Extract Class refactoring prepares most of the changes in the checkFinalConditions() method by calling the performFieldRewrite() and updateReferences() methods. However in the createChange() method, instead of just collecting the changes from the change manager, some more changes are created by calling the createParameterObject() method. This is the method that creates the new class, either top level or nested in Source.

### 4.4.1 Preconditions

The preconditions that we found for the Extract Class refactoring are:

1. *Class* must have at least one field that can be extracted.
2. *Fields* is not empty.
3. No two fields in *Fields* will be created with the same name in *NewClass*.
4. An attribute with the name of *NewField* does not already exist in *Class*.
5. The fields in *Fields* are not static, transient or volatile.
6. If *NewClass* will be made nested.
   6.1 No class with the name of *NewClass* should already exist in *Class*.
7. If *NewClass* will be made top level.
   7.1 No class with the name of *NewClass* should already exist in the package that contains *Class*.
   7.2 The types of the fields in *Fields* are not private.
   7.3 If the type of a field in *Fields* is protected, then the declaring class of that type should be in the same package as where *NewClass* will come.
4.4.2 Mechanics

The mechanics that we found for the Extract Class refactoring are:

1. If for the destination “Top level class” was chosen, a new class (NewClass) will be created in the same package as Class.

2. If for the destination “Nested class in ‘Class’ ” was chosen, a new class (NewClass) will be declared in Class.

3. The fields in Fields will be moved from Class to NewClass.

4. If the option “Create getters and setters” was chosen.

   4.1 The Fields will be made private in NewClass.

   4.2 For each field in Fields a public getter method and a public setter method will be created in NewClass.

   4.3 All accesses of a field in Fields is changed to an access of NewField followed by a call to the getter of the field.

   4.4 All updates of a field in Fields is changed to an access of NewField followed by a call to the setter of the field.

5. If that option was not chosen.

   5.1 The Fields will be public in NewClass.

   5.2 All accesses of a field in Fields will be changed to an access of NewField followed by an access of the field.

   5.3 All updates of a field in Fields will be changed to an access of NewField followed by an update of the field.

6. A new field (NewField) of type NewClass will be created in Class.

   6.1 If at least one field in Fields is public, NewField will be public.

   6.2 If no fields in Fields are public but one or more fields is protected, then NewField will be protected.

   6.3 If no fields in Fields are public or protected, but one or more fields has visibility default, then NewField will get visibility default.

   6.4 If all fields in Fields are private, then NewField will be private.
4.5 Conclusions

In this chapter we have studied how the preconditions and mechanics of a refactoring can be extracted from its implementation in eclipse. For the preconditions we need to read the source code for the methods `checkInitialConditions()` and `checkFinalConditions()`. Anywhere any of the methods `RefactoringStatus.addFatalError()`, `RefactoringStatus.addWarning()` or `RefactoringStatus.addError()` was called one of the preconditions was violated. We can interpret which precondition was violated from the message that is given as a String parameter and from the surrounding if-statements. To find the implemented mechanics we can most easily work backwards, starting from the change that is returned by the `createChange()` method and working back to the change manager object and all the changes/rewrites that are added to this manager.

The preconditions that were found include checks on an existing class, method or field as well as some simple checks on strings. The mechanics include changing existing fields, creating a class, a method or a field, moving a method or a field and removing methods. As the preconditions and mechanics of the three selected refactorings have enough variance we confirm that the three refactorings presented here are indeed a representative selection of the refactorings implemented in eclipse.
In this chapter we will give a formal representation of the selected refactorings, by specifying the preconditions and the mechanics that we found in the eclipse code (see chapter 4) as story diagrams (explained in chapter 3).

The java source code elements (e.g. packages, classes, methods, ....) are represented as a graph, using the meta-model in appendix A. This way we can write the preconditions as subgraphs that may or may not appear in the java program graph and the mechanics can be written as graph transformations.

The story diagram notation uses a different input mechanism than the one in eclipse. In eclipse the input was entered in two phases, first by selecting an element in the editor or the source code views, secondly by entering additional input in the refactoring wizard dialog. In the story diagrams however we will enter all input at once as parameters. Therefore in the story diagrams we need to add some extra preconditions that were implicitly satisfied in the eclipse implementation. For instance in the Pull Up refactoring the dialog gave the option of pulling up the selected method only to a superclass. Whereas in the story diagrams, we need to explicitly check that the parameter \texttt{target} is a superclass of the parameter \texttt{source}. 

5.1 Pull Up Method

5.1.1 Preconditions in Story Diagrams

In this section we give a detailed explanation of the story patterns in the story diagram `checkPreconditions`. The story diagram is shown in figure 5.1. The story patterns are numbered in the upper left corner, this is not part of the story diagram syntax, but merely for convenient referring in this section.

The parameters given to the `checkPreconditions` story diagram are:

- **source** (of type Class) is the class from which the method will be pulled up.
- **target** (of type Class) is the class to which the method will be moved.
- **method** (of type MethodBody) is the method that will be pulled up.
- **makeAbstract** (a boolean) represents whether the method will be declared abstract in **target**.
- **deleteMethods**, a set that contains all the methods in subclasses of **target** that will also be removed.

The return type of this story diagram is boolean. If all preconditions are satisfied it returns `true`, whereas if any of the preconditions are violated it returns `false`.

This story diagram is a specification for the preconditions that we listed in section 4.2.1. With the exception of the checks that **source** is not an enumeration (precondition 1), an annotation (precondition 2), binary (precondition 4) or read-only (precondition 5) since these concepts are not represented in our meta-model. Precondition 3 is satisfied by the fact that the parameter **source** is of type Class.

The added preconditions due to the different input mechanism are:

17. **target** is a superclass of **source**

18. **method** belongs to **source**

19. each method in **deleteMethods** belongs to a subclass of **target** and has the same signature as **method**

In story pattern 1 we check a couple things: We check that **target** is a superclass of **source** (precondition 17); We check that **method** belongs to **source**
Figure 5.1: Complete story diagram to check the preconditions of the Pull Up Method refactoring.
We check that method is not a constructor for source (precondition 9); And we check that neither source nor target are named “Object” (preconditions 6 and part of 7). Note that in order for this last check to be complete we should also check that if either source or target are name “Object” that it does not belong to the package java.lang. This is easily done by adding another story pattern to the diagram, but for the sake of simplicity we left this out and made this precondition a little bit stronger by prohibiting the use of “Object” as the name for the classes.

This story pattern also implicitly checks precondition 7. This was one of the initial preconditions that were checked in eclipse in order to list at least one superclass in the dialog. In our story diagram both the source and the target were already given and we check that they are related by one or more generalizations (superclass*) (i.e. precondition 18).

By using non optional objects for source, target and method we check that these exist and are not null (precondition 8). We also state that the return type of the method (if there is one) is allowed to be source or target.

Story pattern 2 verifies two preconditions from the list in section 4.2.1. Specifically it checks that the skipped classes do not implement a method with the same signature as method (precondition 15) and by stating that the skipped class may be target this also checks that target does not already declare a method with the same signature as method (precondition 11).

In contrast to the first story pattern, we could regard this as a negative preconditions. This means that the precondition checks fail if this pattern is evaluated to true, whereas in the first pattern the preconditions failed if the pattern was evaluated to false (remember from chapter 3 that a story pattern is in fact a complex boolean condition). To check that the signatures are the same we do a String comparison using the signature story diagram that can be found in the appendix B.1.1

The third part of our story diagram (i.e. patterns 3 through 7) deals with the methods in subclasses of target and the preconditions on these methods (preconditions 12, 13, 14 and 19).

To do this we use a “For Each”-Activity (shown in pattern 3), which we use to loop over all methods in all subclasses of target. If we find a method in one of the subclasses that has the same name as method, we follow the arrow labeled “[each time]” to a couple of branches. The first branch is to check if the selected method is one of the methods in the set deleteMethods. If that was the case the signature of that method should be the same as the signature of method (this is checked in pattern 4). In eclipse this precondition
check was not needed, since only methods with the same signature could be selected for deletion anyway (see the dialog in figure 4.3). The second branch leads us to the checks in patterns 5 through 7, where we check the visibility and the return type of the method, but only if the signature of the method is the same as the signature of method. In pattern 5 we verify that the return type of the method is the same as the return type of method and we check that the visibility is not private. In pattern 6 we see if the method belongs to a class which is in a different package than target. If that is the case we proceed to pattern 7 where we verify that the method’s visibility is not default.

Once the “For-Each”-activity finished its loops we come to the final checks (i.e. patterns 8 through 12). If target will be made abstract we come in pattern 12, where we check that target is not instantiated anywhere (precondition 16). We recognise that it is instantiated, if a variable has the target class as type and it is updated with an instantiation expression.

On the other hand, if target will not be made abstract, we need to check that all methods, fields and types that are used in the body of method are accessible from target (precondition 10). This is done by first iterating over all expressions in the body of the method (pattern 8) and then checking if the fields and methods it accesses are accessible by the target using the checkAccesses story diagram in figure B.2 in the appendix (pattern 9). Secondly we iterate over all local variables in the method body (pattern 10) and check if their types are visible from target using the checkTypes story diagram in figure B.3 in the appendix (pattern 11).

## 5.1.2 Mechanics in Story Diagrams

Having covered the preconditions we now give the story diagrams that cover the mechanics of the Pull Up Method refactoring. The mechanics that we identified in eclipse’s implementation are listed in section 4.2.2. The first story diagram (figure 5.2) shows the perform operation. This has the same parameters as the checkPreconditions story diagram. It starts by binding an operation to the method and subsequently checking the preconditions. If this succeeds, we split into two options either we pull up the method (change 2) or we declare an abstract copy in the destination class (change 1).

If makeAbstract is true, we create a copy of method in target and declare it abstract (change 1.1 and 1.2). The copyOperation story diagram can be found in the appendix, in figure B.4. Then we call the makeAbstract story diagram, which is shown in figure 5.3. What this does is make target abstract and subsequently create a method stub (i.e. a method with an empty body) in all subclasses that are not abstract, do not yet implement the method and
5.1. PULL UP METHOD

Figure 5.2: The story diagram that shows the mechanics of the Pull Up Method refactoring
have an abstract class as a direct superclass (change 1.3). This is done in a few steps. The first story pattern makes \texttt{target} abstract. The second story pattern is a “For-Each”-activity that iterates all subclasses of \texttt{target} that are not abstract and have an abstract parent class. The third story pattern checks if the current subclass of \texttt{target} already has a method with the same signature as \texttt{method}. If it already has such a method no stub needs to be created and it returns back to the subclass iteration. If it did not find such a method the fourth story pattern creates a copy of \texttt{method} in the subclass and gives it an empty body. When the makeAbstract story diagram finishes we return to the \texttt{perform} story diagram.

Figure 5.3: The story diagram that makes target class abstract and creates a method stub in all its non abstract subclasses.

If \texttt{makeAbstract} was false, we enter the actual pull up operation. In this story pattern we encounter a graph transformation. We mentioned in chapter 3 that the left-hand side of the transformation can be interpreted by taking the black elements (i.e. the objects \texttt{method}, \texttt{target} and \texttt{source}) and the red elements (i.e. the link between the objects \texttt{method} and \texttt{source}). The right-hand side can be interpreted as the black and the green element (i.e. the objects and the link between the objects \texttt{method} and \texttt{target}). So what it does is remove the link between the objects \texttt{method} and \texttt{source} and create a new link between
the objects method and target. That way method is moved from source to target (change 2.1).

Next it will delete all the methods that were marked for deletion (i.e. all the methods in the set deleteMethods)(change 2.3). To do this we need to re-link all calls to the methods in deleteMethods to method which was moved to target. These calls will remain valid, because target was a superclass to the classes that declared the methods in the set deleteMethods. When the calls are relinked we can delete the methods and its parameters and subsequently delete the method body and recursively remove all expressions that were contained in the method body. This last step is done by calling the destroy story diagram in figure B.5, which can be found in the appendix.

If method was an abstract method (i.e. op.isIsAbstract() == true) then target will be made abstract and stub methods will again be created in the subclasses where necessary (change 2.2). This is done by calling the makeAbstract story diagram in figure 5.3 again. When the stubs are created the methods in the set deleteMethods still need to be removed. Note that this might lead to compilation errors. Indeed the deletion of the method in the subclasses will mean that some subclasses do not implement the inherited abstract method. However this is also how the refactoring works in the implementation in eclipse. A better way would be to first remove the methods in deleteMethods and then create the necessary method stubs.

There is a caveat in these story diagrams (more precisely in the diagrams in figures 5.1 and 5.2). Instead of iterating over the set deleteMethods, we iterate over the methods in the subclasses of target and then we put a branch that checks if the current method is contained in the set deleteMethods. So we do not actually verify that every method in deleteMethods is indeed a method of a subclass of target. But if that set does contain a method that does not belong to a subclass of target, then it will not be removed either. Therefor the story diagrams work, but it is not really how it should be. According to Albert Zündorf [30] a way to invoke a method on a set of nodes (i.e. call the method on each of the nodes, one after the other) is still to be defined. That is why we used the workaround of iterating all nodes and then checking if the current node is in the set.
5.2. ENCAPSULATE FIELD

5.2 Encapsulate Field

5.2.1 Preconditions in Story Diagrams

In this section we cover the story diagram for the precondition checks of the Encapsulate Field refactoring shown in figure 5.4. The story patterns are again numbered in the upper left corner for convenient referring.

The parameters given to the checkPreconditions story diagram are:

- **container** (of type Class) is the class that declares the field that will be encapsulated.
- **var** (of type Variable) is the field (declared in container) that will be encapsulated.
- **getterName** (of type String) the name we want to use for the getter for var.
- **setterName** (of type String) the name we want to use for the setter for var.

The return type of this story diagram is boolean. The return value indicates whether or not all preconditions were satisfied.

This story diagram is a specification for the preconditions that were listed in section 4.3.1. There is only one added precondition check due to the different input mechanism:

8. **var** is declared in **container**.

Story pattern 1 starts with some basic precondition checks. It verifies that the type of **var** exists (precondition 1) and that **var** belongs to **container** (precondition 8). By calling the checkValidJavaIdentifier story diagram in figure B.6, it also verifies that the setterName and getterName strings are not the empty string (precondition 2) and that they are a valid Java identifier (precondition 3).

Story pattern 2 does not do any precondition checks, but merely searches for an existing getter. A getter can be recognised as a method that has as a name **getterName** and has no parameters. Note the use of a negative object to indicate that the method has no parameters. If a getter already existed then the transition to story pattern 3 would be followed, where we check that the existing getter has as a return type the type of **var** (precondition 5) and that it has the same static property as **var** (the getter part of precondition 4).
5.2. ENCAPSULATE FIELD

Figure 5.4: The story diagram to check the preconditions of the Encapsulate Field refactoring.
If a getter did not yet exist then a new one will be created in the refactoring and we need to check that this will not override a method in a superclass (the getter part of precondition 7). This is done in story pattern 4, where we look in all superclasses of container for a method that has as a name `getterName` and has no parameters.

Story pattern 5 is similar to 2, in that this one searches for an existing setter, which is recognised as a method that has as a name `setterName` and has a single parameter that has as type, the type of `var`. If a setter is found, we continue to story pattern 6, where we check that the existing setter has the same static property as `var` (the setter part of precondition 4) and that the setter has no return type (precondition 6).

If a setter was not found then we need to check that the one that will be created will not override any method in a superclass (the setter part of precondition 7). This check is done in story pattern 7.

### 5.2.2 Mechanics in Story Diagrams

In this section we show the story diagrams that cover the mechanics of the Encapsulate Field refactoring. The mechanics that we identified in eclipse's implementation are listed in section 4.3.2. The first story diagram (figure 5.5) shows the `perform` operation, which has two more parameters than the `checkPreconditions` story diagram.

- **useAccessors** (of type boolean) If set to true, the accesses and updates of `var` inside `container` will also be converted to calls to the getter and setter methods.

- **accessorVisibility** (of type String) is the visibility the user requests for the getter and setter methods.

This story diagram starts with storing the visibility of `var` in a local temporary variable. This is needed later on when determining what the visibility of the getter and setter methods will be. In the second story pattern we change the visibility of `var` to private (change 1). And in the third story pattern we call two story diagrams that deal with the creation of the getter and setter methods. In this section we will only cover the creation of the getter method. The story diagrams dealing with the creation of the setter method can be found in the appendix (figures B.7 and B.8).

The `createGetter` story diagram (shown in figure 5.6) deals with change 2. This diagram starts by looking for an existing getter in `container`. If it did not find one we will create one. The new getter method will get as name
5.2. ENCAPSULATE FIELD

getterName, will be static if var was static and its visibility is determined by accessorVisibility and the visibility var was before the operation. The visibility for the getter is determined in the story diagram in figure B.9. The body of the getter method looks like: “return var;”. This is represented in graph notation by a return node, that contains an access node, which is linked to var.

The last story pattern calls the encapsulateGetter story diagram for each access that is linked to var. With the exception of the access inside the getter method. This encapsulateGetter story diagram is shown in figure 5.7. In this story diagram we handle changes 3 and 5. If useAccessors is false, it will first check that the access we need to convert is indeed in a class different from container. The conversion is done by removing the access expression and creating in its place a call expression, that is linked to the getter.
Figure 5.6: The story diagram to create a getter in the Encapsulate Field refactoring.
Figure 5.7: The story diagram that changes field accesses to getter calls in the Encapsulate Field refactoring.
5.3. Extract Class

5.3.1 Preconditions in Story Diagrams

In this section we cover the story diagram for the precondition checks of the Extract Class refactoring shown in figure 5.8. The story patterns are again numbered in the upper left corner for convenient referring.

The parameters given to the checkPreconditions story diagram are:

- **source** (of type Class) the class from which we want to extract the new class.
- **fields** a set that contains the fields declared in source, that we want to move to the new class.
- **fieldName** (of type String) the name we want to use for the new field in source.
- **newClassName** (of type String) the name we want to use for the new class.
- **nested** (a boolean) indicating whether we want to create the new class top level or nested in source.

The return type of this story diagram is boolean. The return value indicates whether or not all preconditions were satisfied.

This story diagram is a specification for the preconditions that were listed in section 4.4.1. Precondition 3 is not represented because in our story diagram implementation we do not allow the renaming of the fields. We will also not check that the fields in fields are not transient or volatile (part of precondition 5) since these concepts are not available in the meta-model.

There is only one added precondition check due to the different input mechanism:

8. Each field in fields is declared in source.

Story pattern 1 starts with some basic precondition checks. It makes sure that each field in fields is declared in source (precondition 8). Additionally we verify that each of those fields is not static (precondition 5). By using a boolean constraint (“{!fields.isEmpty()}”) we check preconditions 2 and 1. We use a negative object in the story pattern to verify that source does not already contain a field with the name fieldName (precondition 4).
Figure 5.8: The story diagram that checks the preconditions for the Extract Class refactoring.
After this the checks are split into two cases, one for when \textbf{nested} is true (precondition 6) and one for when it is false (precondition 7). If we want to make the new class nested in \texttt{source}, then we end up in story pattern 2, where we check that there does not already exist a class with the name \texttt{newClassName} in \texttt{source} (precondition 6.1).

If we want to make the new class top level, then we have to check that no class with the name \texttt{newClassName} already exists in the package that contains \texttt{source} (precondition 7.1). Next we loop over all fields in \texttt{fields} in order to check the visibility of their types. In story pattern 5 we check that the types of the fields in \texttt{fields} are not private (precondition 7.2). Further down, in pattern 6, we check that if the types are protected that they are declared in the package that contains \texttt{source} (i.e. the package where the new class will be created)(precondition 7.2).

5.3.2 Mechanics in Story Diagrams

In this section we show the story diagram that covers the mechanics of the Extract Class refactoring. The mechanics that we identified in eclipse’s implementation are listed in section 4.4.2. This story diagram (figure 5.9) shows the \textit{perform} operation, which has the same parameters as the \textit{checkPreconditions} story diagram. We have chosen for simplicity not to make the choice “Create getters ans setters” optional. Therefore in this story diagram we will not cover change 5.

The first story pattern already deals with a lot of the changes. We create a new Class (\texttt{newClass}) and set its name to \texttt{newClassName}. The fields in the set \texttt{fields} are moved from \texttt{source} to \texttt{newClass} (change 3) and the visibility of all those fields is set to private (change 4.1). This story pattern also creates a new variable \texttt{newField} of type \texttt{newClass} and places it in \texttt{source} (change 6. The name of this new variable is set to \texttt{fieldName} and its visibility is determined by the \textit{getHighestVisibility} story diagram in figure B.10.

Now if \texttt{nested} is true, we place \texttt{newClass} in \texttt{source} (change 2). Else we place it in the package that contains \texttt{source} (change 1). At the end of this story diagram we call the \textit{createGettersAndSetters} story pattern for each of the fields that was moved to \texttt{newClass} which obviously creates a getter and a setter method for each field and also converts all accesses and updates of the fields to an access of \texttt{newField} followed by a call to the respective getter or setter (changes 4.2, 4.3 and 4.4). This is done analogously to the Encapsulate Field refactoring and can be found in the appendix in figures B.11, B.12 and B.13.
Figure 5.9: The story diagram that shows the mechanics of the Extract Class refactoring.
5.4 Conclusions

As we have shown in this chapter specifying refactorings in the form of story diagrams is indeed possible. The more sophisticated features of Fujaba such as negative and optional nodes and edges, path expressions, for-each activities are useful in the creation of understandable rules. Additionally the possibility to model complex control flow and to extract subfunctionality to different story diagrams help to formalise these rather complex refactorings.

When specifying the story diagrams we needed to introduce extra preconditions to account for the different input mechanism. In the story diagrams all input is given at once in the form of parameters, whereas in eclipse the first input is a textual selection in the source code followed by some more input in the refactoring wizards. The input that can be given in the refactoring wizard in eclipse is dependant on the initial input. For instance in the Pull Up Method refactoring, only a class that is a superclass of the declaring class could be selected as destination class. In the story diagrams however both the declaring class and destination class are given as a parameter, therefore we needed to add an extra precondition that the destination class needs to be a superclass of the declaring class.

This different input mechanism might also have effect on the specifications of refactorings that, in eclipse, needs to work on a large text selection. Take for instance the Extract Method refactoring. This is a refactoring in which a series of statements and expressions in a method are replaced by a call to a new method that embodies the selection. If we were to specify this using story diagrams we would have to add the precondition check that the order of the given expressions and statements is correct (e.g. that no expressions are missing, ...).

Another issue is that the metamodel that we used does not represent all possible java constructs. Consequently we can not express all the preconditions in our story diagrams. However this doesn’t harm our experiment, since the refactoring can still work without these constructs. It just means that we have less cases to cover in our tests.
CHAPTER 6

Validation of the Specifications

In figure 1.1 we showed an overview of the validation process, with which we will test the story diagrams that we created. What this boils down to is that we will execute the story diagrams and see if the resulting java program graph is what we expect it to be: a graph representation of the java code after executing the refactoring in eclipse. The story diagrams can be executed as we created them using Fujaba, which uses the CodeGen2\cite{14} engine for its code generation. This converts the elements in the meta-model (appendix A) into classes. The nodes in a java program graph are then instances of such a class. The code generation also converts each of the story diagrams into a java method, that we can call from our testing framework. To visualise the java program graphs we use eDOBS\cite{2}, an eclipse plugin, that visualizes the current heap of a Java program at runtime as a UML object diagram.

Since there is no tool support for the translation step (i.e. translate the java code into a java program graph) we have to do this step manually. Therefore we only work with very small sample code.

In the following sections we show the visualised java program graphs before and after executing the refactoring. However as this is a tedious and time-consuming way of verifying the refactorings we only do this for the main scenarios as an illustration. The rest of the scenarios of the refactoring as well as the preconditions are tested using JUnit\cite{3}. Each of the preconditions is

\footnote{1\text{http://www.se.eecs.uni-kassel.de/se/index.php?codegen2}}

\footnote{2\text{http://www.se.eecs.uni-kassel.de/se/index.php?edobs}}

\footnote{3JUnit is a simple framework for automated testing of java code.}
tested separately by manipulating the java program graph in such a way that we know the precondition should fail and all other preconditions are satisfied.

### 6.1 Testing Pull Up Method

#### 6.1.1 Main Scenarios

To test the Pull Up Method refactoring we use the sample test code in listing 6.1. This is a simple class hierarchy so we can pull up a method. The source class (i.e. the class from which we will pull up a method) has a few sibling classes that declare a “similar” method (i.e. a method with the same signature), which can be removed. The java program graph that we translated this code into is visualised in the graph in figure 6.1. The mapping between the java code and the nodes in the java program graph, as visualised by eDOBS, is shown in table 6.1.

We will now pull up the method method from the class source to the class grandFather and at the same time remove the methods sibling2.method() and sibling3.method(). Performing this refactoring in eclipse, results in the code in listing 6.2.

To perform the refactoring as it is defined in the story diagrams we need to instantiate an object of type PullUpMethodRefactoring (generated by Fujaba), which declares the methods that are generated from the story diagrams. On this PullUpMethodRefactoring object we can now call the method perform(c4, c1, m26, false, {m36, m32}). The parameters given here are the identifiers of the nodes in the graph: c4 is the class that declares the method m26; c1 is the class to which we want to pull up the method; false indicates we do not want to declare the method abstract in c1 and the set {m36, m32} is the set of methods we want to remove. The call to the perform method results in the graph visualised in figure 6.2. As we can see the link between the nodes c4 and m26 is removed, and a new link was created between the nodes c1 and m26. Additionally the nodes m36, o37, b38, m32, o34 and b35 have been removed (these represented the methods sibling2.method() and sibling3.method()). We can also easily see that this resulting graph is a graph representation for the code we got when performing the refactoring in eclipse (i.e. listing 6.2).

Similarly we also perform the refactoring with the “declare abstract in destination” action. This means that the class grandFather will be made ab-
abstract and will declare an abstract version of the method `source.method()`.
The subclasses `parent` and `uncle` will declare a method stub conforming to
the newly created abstract method. The resulting code is shown in listing 6.3.

For the story diagram version of the refactoring we have to call the method
`perform(c4, c1, m26, true, {})` on the `PullUpMethodRefactoring` object.
The result of this is the graph shown in figure 6.3. As we can see the methods
in `source` and its sibling classes have remained where they were. A method
without a body was declared in `grandFather` (i.e. an abstract method) and a
method was added to both the classes `parent` and `uncle`. This is, as we would
expect, a graph representation for the code in listing 6.3, which was the code
we got after performing the refactoring in eclipse.
Table 6.1: Mapping between the java code in listing 6.1 and the nodes in the graph in figure 6.1

<table>
<thead>
<tr>
<th>Node Identifiers</th>
<th>Java Code They Represent</th>
</tr>
</thead>
<tbody>
<tr>
<td>p0</td>
<td>package testPullUpMethod.family</td>
</tr>
<tr>
<td>p33</td>
<td>package testPullUpMethod.test</td>
</tr>
<tr>
<td>c17</td>
<td>public class Integer() (aka int)</td>
</tr>
<tr>
<td>c1</td>
<td>public class grandFather{}</td>
</tr>
<tr>
<td>c2</td>
<td>public class parent extends grandFather{}</td>
</tr>
<tr>
<td>c3</td>
<td>public class uncle extends grandFather{}</td>
</tr>
<tr>
<td>c4</td>
<td>public class source extends parent{</td>
</tr>
<tr>
<td></td>
<td>public void method(){</td>
</tr>
<tr>
<td></td>
<td>int i;</td>
</tr>
<tr>
<td></td>
<td>i = i + 1;</td>
</tr>
<tr>
<td></td>
<td>i = 0;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>c5</td>
<td>public class sibling1 extends parent{</td>
</tr>
<tr>
<td></td>
<td>private int i;</td>
</tr>
<tr>
<td></td>
<td>private void foo(){</td>
</tr>
<tr>
<td></td>
<td>public void method(){</td>
</tr>
<tr>
<td></td>
<td>i = 0;</td>
</tr>
<tr>
<td></td>
<td>foo();</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>c6</td>
<td>public class sibling2 extends parent{</td>
</tr>
<tr>
<td></td>
<td>public void method(){</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>c7</td>
<td>public class sibling3 extends parent{</td>
</tr>
<tr>
<td></td>
<td>public void method(){</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>
Listing 6.1: Java code for the Pull Up tests (conforming to the graph in figure 6.1).

```java
package testPullUpMethod.family;
public class GrandParent {}

package testPullUpMethod.family;
public class Parent extends GrandParent {

package testPullUpMethod.family;
public class Uncle extends GrandParent {

package testPullUpMethod.family;
public class Source extends Parent {
    public void method() {
        int i;
        i = i + 1;
        i = 0;
    }
}

package testPullUpMethod.family;
public class Sibling1 extends Parent {
    public int i;
    public void foo() {
        public void method() {
            i = 0;
            foo();
        }
    }
}

package testPullUpMethod.family;
public class Sibling2 extends Parent {
    public void method() {
    }
}

package testPullUpMethod.test;
import testPullUpMethod.family.Parent;
public class Sibling3 extends Parent {
    public void method() {
    }
}
```
6.1. TESTING PULL UP METHOD

Figure 6.1: Java Program Graph representing the java code in listing 6.1
Listing 6.2: Java code after pulling up method from Source and removing the similar method from Sibling2 and Sibling3 (corresponding to the graph in figure 6.2).

```java
1 package testPullUpMethod.family;
2 public class GrandParent {
3     public void method() {
4         int i;
5         i = i + 1;
6         i = 0;
7     }
8 }

9 package testPullUpMethod.family;
10 public class Parent extends GrandParent {}

11 package testPullUpMethod.family;
12 public class Uncle extends GrandParent {}

13 package testPullUpMethod.family;
14 public class Source extends Parent {}

15 package testPullUpMethod.family;
16 public class Sibling1 extends Parent {
17     public int i;
18     public void foo() {} 
19     public void method() {
20         i = 0;
21         foo();
22     }
23 }

24 package testPullUpMethod.family;
25 public class Sibling2 extends Parent {
26 }

27 package testPullUpMethod.test;
28 import testPullUpMethod.family.Parent;
29 public class Sibling3 extends Parent {
30 }
```
Figure 6.2: Java Program Graph after performing the story diagram
PullUpRefactoring.perform(c4, c1, m26, false, \{m36, m32\})
Listing 6.3: Java code after making the method abstract in GrandParent (corresponding to the graph in figure 6.3).

```java
package testPullUpMethod.family;
public abstract class GrandParent {
    public abstract void method();
}

package testPullUpMethod.family;
public class Parent extends GrandParent {
    public void method() {} ...
}

package testPullUpMethod.family;
public class Uncle extends GrandParent {
    public void method() {} ...
}

package testPullUpMethod.family;
public class Source extends Parent {
    public void method() {
        int i = 1;
        i = i + 2;
    }
}

package testPullUpMethod.family;
public class Sibling1 extends Parent {
    public int i;
    public void foo() {} ...
    public void method() {
        i = 0;
        foo();
    }
}

package testPullUpMethod.family;
public class Sibling2 extends Parent {
    public void method() {} ...
}

package testPullUpMethod.test;
import testPullUpMethod.family.Parent;
public class Sibling3 extends Parent {
    public void method() {} ...
}
```
Figure 6.3: Java Program Graph after performing the story diagram 
\texttt{PullUpRefactoring.perform(c4, c1, m26, true, {})}
6.1. TESTING PULL UP METHOD

6.1.2 Further Testing

To make the testing of the story diagrams easier and repeatable, we use the JUnit testing framework. We use assertions to verify that the results of the story diagrams are as we expect them to be, i.e. that they do the same as what we learnt from the eclipse executions.

Testing the preconditions is done one by one. We make sure that only one of the preconditions is violated in each test. Each precondition is tested by firstly manipulating the java program graph, that was shown in figure 6.1, in such a way that the precondition is violated. And secondly we verify that the method checkPreconditions returns false.

Take for example the code in listing 6.4. This shows some of the tests used to test preconditions of the Pull Up Method refactoring. The first testcase (public void testSourceHasNoSuperclass()) manipulates the java program graph in such a way that the class source has no superclasses. When we call the checkPreconditions method we expect it to return false, since the target class can not be a superclass to the source class.

The last case (public void testTargetAlreadyHasMethod()) changes the graph so that the class grandfather already has a method similar to the method that we want to pull up. The result of calling the checkPreconditions method should also be false.

Listing 6.4: Extract of the source code that test the preconditions for the Pull Up Method refactoring.

```java
public void testSourceHasNoSuperclass()
{
    source.removeAllFromSuperclass();
    assertFalse(checkPreconditions(source, grandfather, m1, false, new HashSet()));
    assertFalse(checkPreconditions(source, parent, m1, false, new HashSet()));
}

public void testTargetNotSuperclass()
{
    assertFalse(checkPreconditions(source, uncle, m1, false, new HashSet()));
}

public void testPullUpNotOwnedMethod()
{
    assertFalse(checkPreconditions(source, grandfather, m5, false, new HashSet()));
}

public void testTargetAlreadyHasMethod()
{
    m5.setClass(grandfather);
    assertTrue(classContainsMethodWithName(grandfather, methodName));
    assertFalse(checkPreconditions(source, grandfather, m1, false, new HashSet()));
}
```
The mechanics are tested in a similar manner. For example the code in listing 6.5 shows one of those tests. In this test, we manipulate the graph so that the method in the class source is abstract (m1.getOperation().setIsAbstract(true)). For this to be complete we should also remove the methodbody from the method, however this won’t affect the outcome of the story diagram. After performing the refactoring, the classes that declare a method with the same signature as the method that was pulled up have changed. In other words, where before the refactoring classes parent, grandFather and uncle did not declare the method, they do after. And the classes sibling2 and sibling3 have had their declaration of the method removed.

Listing 6.5: Extract of the source code that test the mechanics for the Pull Up Method refactoring.

```
665  public void testPullUpAbstractMethod(){
666      HashSet h = new HashSet();
667      h.add(m2);
668      h.add(m5);
669      m1.getOperation().setIsAbstract(true);
670      assertTrue(classContainsMethodWithName(source, methodName));
671      assertTrue(classContainsMethodWithName(sibling1, methodName));
672      assertTrue(classContainsMethodWithName(sibling2, methodName));
673      assertTrue(classContainsMethodWithName(sibling3, methodName));
674      assertFalse(classContainsMethodWithName(grandFather, methodName));
675      assertFalse(classContainsMethodWithName(parent, methodName));
676      assertFalse(classContainsMethodWithName(uncle, methodName));
677      ref.perform(source, grandFather, m1, false, h);
678      assertTrue(grandFather.isIsAbstract());
679      assertFalse(source.hasInMethodBody(m1));
680      assertTrue(grandFather.hasInMethodBody(m1));
681      assertFalse(sibling2.hasInMethodBody(m2));
682      assertFalse(sibling3.hasInMethodBody(m5));
683      assertTrue(sibling1.hasInMethodBody(m4));
684      assertFalse(classContainsMethodWithName(source, methodName));
685      assertTrue(classContainsMethodWithName(sibling1, methodName));
686      assertFalse(classContainsMethodWithName(sibling2, methodName));
687      assertFalse(classContainsMethodWithName(sibling3, methodName));
688      assertTrue(classContainsMethodWithName(grandFather, methodName));
689      assertFalse(classContainsMethodWithName(parent, methodName));
690      assertTrue(classContainsMethodWithName(uncle, methodName));
691  }
```

Other tests for the mechanics of the Pull Up Method refactoring include: a test where the method is pulled up and the methods in sibling2 and sibling3 are removed; a test where the method is pulled up and the method in sibling1 is removed; a test where the method is pulled up and no sibling methods were removed and two tests on declaring the method abstract in the class grandFather.
6.2 Testing Encapsulate Field

6.2.1 Main Scenario

The tests for the Encapsulate Field refactoring are done on the code in listing 6.6. The java graph representation for this code as viewed with eDOBS is shown in figure 6.4. To clarify the graph, we have mapped the nodes to the code in table 6.2.

<table>
<thead>
<tr>
<th>Node Identifiers</th>
<th>Java Code They Represent</th>
</tr>
</thead>
<tbody>
<tr>
<td>p0</td>
<td>package testEncapsulateField;</td>
</tr>
<tr>
<td>c14</td>
<td>public class Integer() (aka int)</td>
</tr>
<tr>
<td>c1</td>
<td>public class class1{</td>
</tr>
<tr>
<td></td>
<td>public void foo(){</td>
</tr>
<tr>
<td></td>
<td>class2 c = new class2();</td>
</tr>
<tr>
<td></td>
<td>c.i = 9;</td>
</tr>
<tr>
<td></td>
<td>int tmp = c.i;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>c2</td>
<td>public class class2{</td>
</tr>
<tr>
<td></td>
<td>public int i;</td>
</tr>
<tr>
<td></td>
<td>public void method(){</td>
</tr>
<tr>
<td></td>
<td>i = i + 1;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

We execute the Encapsulate Field refactoring on the public attribute i of the class class2. We use the option of keeping the field references in the declaring class (i.e. the getter and setter will only be used in class1). The resulting code of performing this refactoring in eclipse is presented in listing 6.7.

To perform the refactoring on the java program graph, we need to create an instance of the EncapsulateFieldRefactoring class. This object can then be used to perform the story diagrams specified in chapter 5. The method called is then EncapsulateFieldRefactoring.perform(c2, v4, ‘‘getI’’, ‘‘setI’’, false, ‘‘public’’), with parameters: c2 to indicate in which class we will encapsulate a field, v4 as the field that we want to encapsulate,
“getI” and “setI” as the names for the getter and setter, false to indicate we do not want to use the getter and setter in the class c2 and “public” for the visibility of the getter and setter. The result is the java program graph in figure 6.5. As can be seen in this graph two methods (the getter and setter) have been added to class c2. The access node (a13) that was linked to the variable node v4 has been replaced by a call node (c32 that is linked to the getter operation (o33). Similarly the update node (u17) has been changed into a call node (31) linked to the setter operation node (o38).

Listing 6.6: Java code for the Encapsulate Field tests (corresponding to the graph in figure 6.4).
```java
1 package testEncapsulateField;
2 public class class1 {
3     public void foo() {
4         class2 c = new class2();
5         c.i = 9;
6         int tmp = c.i;
7     }
8 }
```

Listing 6.7: Java code after Encapsulating i in class2. (corresponding to the graph in figure 6.5).
```java
1 package testEncapsulateField;
2 public class class2 {
3     private int i;
4     public void method() {
5         i = i + 1;
6     }
7 }
```
```java
1 package testEncapsulateField;
2 public class class2 {
3     private int i;
4     public void method() {
5         i = i + 1;
6     }
7     public void setI(int i) {
8         this.i = i;
9     }
10     public int getI() {
11         return this.i;
12     }
13 }
```
Figure 6.4: Java Program Graph representing the java code in listing 6.6
Figure 6.5: Java Program Graph after performing the story diagram `EncapsulateFieldRefactoring.perform(c2, v4, "getI", "setI", false, "public")`
6.2.2 Further Testing

As with the Pull Up Method refactoring, the tests are written in the JUnit framework. The tests start with the java program graph that was shown in figure 6.4 and then change this graph in a small way. The precondition tests are written in such a way that only one precondition is violated in each test to verify that that precondition violation is detected by our checkPreconditions story diagram.

In listing 6.8 we show two of the tests we did to test the preconditions. The first one (public void testExistingGetterWithDifferentReturnType()) changes the java program graph so that class2 declares a method with the name getterName and returns a value of type class1. Since our variable (var1) is of type Integer the preconditions should be violated as this is a different type than the return type of the existing getter. In the second test (public void testNewGetterOverridesParentMethod()) we change the java program graph so that a method with the name getterName is declared in the parent class of class2. Now the preconditions should be violated as the new getter that would be created in class class2 will override the method in the superclass.

Listing 6.8: Extract of the JUnit tests to test the preconditions of the Encapsulate Field refactoring.

```java
public void testExistingGetterWithDifferentReturnType(){
makeNewOperationInClass(class2, getterName, class1);
assertFalse(checkPreconditions(class2, var1, getterName, setterName));
}

public void testNewGetterOverridesParentMethod(){
Class parent = new Class();
Operation getter = makeNewOperationInClass(parent, getterName, Integer);
parent.addToSubclass(class2);
parent.addToMethodBody(getter.getMethodBody());
assertFalse(checkPreconditions(class2, var1, getterName, setterName));
}
```

In listing 6.9 we show one of the tests that test the mechanics of the Encapsulate Field refactoring. We start by manipulating the graph a little and give class2 a getter method for var1 (lines 410 to 416). On line 420 we assure that no setter method already exists. The rest of the statements before the perform operation are just to remember what the graph looked like before the execution of the story diagram. Then on line 431 we call the perform operation on the EncapsulateField object (ref). After the execution of the refactoring we perform a few tests on the result. The number of methods in class2 should be increased by 1 (the setter method). When we look for the setter it should be found. The parent and child expressions of the access expression acc2 and update expression up1 should have remained the same, since these are the access and update expressions in class2 itself. Whereas the update expression
up3 should be changed to a call to the setter method as this is an update in the class class1.

Other cases that were tested include: the use of an existing Setter method, a test were both the setter and getter were created, and then the same set of tests again but then with the useAccessors parameter set to true. (i.e. also update the accesses and updates inside the declaring class). Additionnaly we have a number of testcases to test the visibility of the created getter and setter method. (e.g. when the field is private any requested visibility is allowed, but when the field was public the requested visibility is ignored and the setter/getter methods are made public.)

Listing 6.9: Extract of the JUnit tests to test the mechanics of the Encapsulate Field refactoring.

```java
public void testUseExistingGetter() {
    Operation getter = makeNewOperationInClass(class2, getterName, Integer←);
    Block bl = getter.getMethodBody().getBlock();
    Return ret = new Return();
    bl.addToSubExpression(ret);
    acc6.setVariable(var1);
    ret.addToSubExpression(acc6);
    int Oldsize = class2.sizeOfMethodBody();
    MethodBody setter = findMethodInClass(class2, setterName);
    assertNotNull(setter==null);
    Expression parent = up1.getFirstOfSuperExpression();
    Expression child = acc2.getFirstOfSubExpression();
    assertNotNull(parent);
    assertNotNull(child);
    Expression parent2 = up3.getFirstOfSuperExpression();
    Expression child2 = up3.getFirstOfSubExpression();
    assertNotNull(parent2);
    assertNotNull(child2);
    ref.perform(class2, var1, getterName, setterName, false, "public");
    //nr of methods is increased by 1
    int size = class2.sizeOfMethodBody();
    assertTrue(size == Oldsize + 1);
    setter = findMethodInClass(class2, setterName);
    assertNotNull(setter != null);
    //check acc2 and up1 remain the same
    assertTrue(up1.getFirstOfSuperExpression()==parent);
    assertTrue(up1.getFirstOfSubExpression()==acc2);
    assertTrue(acc2.getFirstOfSuperExpression()==up1);
    assertTrue(acc2.getFirstOfSubExpression()==child);
    //Check update 3 became call to setter
    Call c = (Call)exp;
    assertTrue(c.getOperation().getName().equals(setterName));
    assertTrue(c.getActualParameters2().hasInExpression(child2));
}
```
6.3 Testing Extract Class

6.3.1 Main Scenario

The code used for the Extract Class refactoring is shown in listing 6.10. Its java program graph representation as visualised in eDOBS is shown in figure 6.6. The mapping between the nodes and the java code is given in table 6.3.

Table 6.3: Mapping between the java code in listing 6.10 and the nodes in the graph in figure 6.6

<table>
<thead>
<tr>
<th>Node Identifiers</th>
<th>Java Code They Represent</th>
</tr>
</thead>
<tbody>
<tr>
<td>p0</td>
<td>package testExtractClass.test;</td>
</tr>
<tr>
<td></td>
<td>class Integer() (aka int)</td>
</tr>
<tr>
<td>c1</td>
<td>public class class1{</td>
</tr>
<tr>
<td></td>
<td>public int field1;</td>
</tr>
<tr>
<td></td>
<td>public int field2;</td>
</tr>
<tr>
<td></td>
<td>public void foo(){</td>
</tr>
<tr>
<td></td>
<td>field1 = 0;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>c2</td>
<td>public class class2{</td>
</tr>
<tr>
<td></td>
<td>public void boo(){</td>
</tr>
<tr>
<td></td>
<td>class1 c = new class1();</td>
</tr>
<tr>
<td></td>
<td>c.field2 = 2;</td>
</tr>
<tr>
<td></td>
<td>int j = c.field1;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

The main scenario that we want to perform is extracting a class (that we shall call “newClass”) from the class class1. We will move the fields field1 and field2 to this new class. The name of the new field that is added to class1 is “data”. The result of executing this refactoring in eclipse is shown in the code in listing 6.11.

To perform this refactoring on the graph, we have to instantiate an object of the type EncapsulateFieldRefactoring. On this object we can then call the method perform(c1, \{v12,v13\}, ’’data’’, ’’newClass’’, true). The parameters are to indicate that c1 is the class from which we will extract a class and v12 and v13 are the fields that will be moved to the new class. The last parameter is to indicate that now we do want to convert the updates and
accesses in the declaring class to getter and setter calls.

The resulting graph of this refactoring is shown in figure 6.7. And the result is what we expected it to be. A new class is created within c1 and the variables v12 and v13 are moved to this new class and each one of these is provided with a getter and setter method. The updates u17 and u20 have been converted to a call to the right setter method. And the access node a25 is changed into a call to a getter method.
Listing 6.10: Java code for the Extract Class tests (corresponding to the graph in figure 6.6).

```java
package testExtractClass.test;
public class class1{
    public int field1;
    public int field2;
    public void foo(){
        field1 = 0;
    }
}
```

```java
package testExtractClass.test;
public class class2 {
    public void boo(){
        class1 c = new class1();
        c.field2 = 2;
        int j = c.field1;
    }
}
```

Listing 6.11: Java code after extracting field1 and field2 from class1 into a new nested Class. (corresponding to the graph in figure 6.7).

```java
package testExtractClass.test;
public class class1{
    public class newClass{
        private int field1;
        private int field2;
        public void setField1(int field1){
            this.field1 = field1;
        }
        public int getField1(){
            return this.field1;
        }
        public void setField2(int field2){
            this.field2 = field2;
        }
        public int getField2(){
            return this.field2;
        }
    }
    public newClass data = new newClass();
    public void foo(){
        data.getField1(0);
    }
}
```

```java
package testExtractClass.test;
public class class2 {
    public void boo(){
        class1 c = new class1();
        c.data.setField2(2);
        int j = c.data.getField1();
    }
}
```
Figure 6.6: Java Program Graph representing the java code in listing 6.10
Figure 6.7: Java Program Graph after performing the story diagram `ExtractClassRefactoring.perform(c1, {v12,v13}, “data”, “newClass”, true)`
6.3. Testing Extract Class

6.3.2 Further Testing

Again we use JUnit to do more tests on the story diagrams for the Extract Class refactoring. For each of the tests we start with the java program graph of the code in listing 6.10. Each of the preconditions is again tested separately in a test where only one precondition under test is violated.

Listing 6.12 shows two of the preconditions that were tested. In the first one we begin by asserting that the graph to begin with satisfies the preconditions. Next we manipulate the graph so that a nested class with the name newClassName already exists in the class class1. The call to the checkPreconditions method with the same parameters should now return false.

In the second test we start by creating a new private nested class inside class1 that is used as the type for one of the fields. Now the preconditions should fail when we extract a class top level (the parameter makeAbstract is set to false), since it would then not be able to access the type of one of its attributes. However if we extract the class as a nested class in class1 then the preconditions will succeed (the parameter makeAbstract is set to true).

Listing 6.12: Extract of the JUnit tests to test the preconditions of the Extract Class refactoring.

223 public void testNestedClassAlreadyExists(){
224     HashSet h = new HashSet();
225     h.add(field1);
226     h.add(field2);
227     assertTrue(checkPreconditions(class1, h, fieldname, newClassName, true));
228     Class nestedClass = new Class();
229     nestedClass.setName(newClassName);
230     nestedClass.setEnvelopingClass(class1);
231     assertFalse(checkPreconditions(class1, h, fieldname, newClassName, true));
232 }

288 public void testTypeOfFieldIsPrivate(){
289     Class nestedClass = new Class();
290     nestedClass.setName("someClass");
291     nestedClass.setVisibility("private");
292     class1.addToNestedclasses(nestedClass);
293     field2.setClassifier(nestedClass);
294     HashSet h = new HashSet();
295     h.add(field1);
296     h.add(field2);
297     assertTrue(checkPreconditions(class1, h, fieldname, newClassName, true));
298     assertFalse(checkPreconditions(class1, h, fieldname, newClassName, false));
299 }

In listing 6.13 we show one of the tests that test the mechanics of the
6.4. TEST RESULTS

Extract Class refactoring. We start by asserting that the fields we will extract into the new class belong to class1 and have visibility public. After the execution of the refactoring class1 should contain a nested class with the name newClassName. The fields that were extracted are now declared in this new nested class and no longer in the old class. To finish the test we validate that the visibility of the fields was changed to private.

Other tests include validating that the creation of a top level class was done properly, that the old declaring class receives a new attribute that has the new class as a type, that a getter and a setter method is created for each of the fields and that the accesses and updates to the fields are converted accordingly.

Listing 6.13: Extract of the JUnit tests to test the mechanics of the Extract Class refactoring.

```java
382 public void testFieldsMovedToNewClass(){
383     HashSet h = new HashSet();
384     h.add(field1);
385     h.add(field2);
386     assertTrue(class1.hasClassInVariable2(field1));
387     assertTrue(class1.hasClassInVariable2(field2));
388     assertTrue(field1.getVisibility().equals("public"));
389     assertTrue(field2.getVisibility().equals("public"));
390     ref.perform(class1, h, fieldname, newClassName, true);
391     assertTrue(hasNestedClass(class1, newClassName));
392     assertFalse(hasTopLevelClass(pack, newClassName));
393     Class newClass = getNestedClass(class1, newClassName);
394     assertTrue(newClass.hasClassInVariable2(field1));
395     assertTrue(newClass.hasClassInVariable2(field2));
396     assertFalse(class1.hasClassInVariable2(field1));
397     assertFalse(class1.hasClassInVariable2(field2));
398     assertTrue(field1.getVisibility().equals("private"));
399     assertTrue(field2.getVisibility().equals("private"));
400 }
```

6.4 Test Results

In figure 6.8 we show a screenshot of the test results. This shows the jUnit test results and the test coverage, which was measured with eclemma. All tests that we wrote are successfully executed in the JUnit framework. There were

4EclEmma (http://www.eclemma.org/) is a Java code coverage plugin for Eclipse, based on the EMMA Java code coverage tool (http://emma.sourceforge.net/).
75 tests: 31 for Encapsulate Field, 20 for Extract Class and 24 for Pull Up Method. As we can see from the coverage measurement, we tested an average of 85% of our story diagrams.

This means that our story diagrams are tested rather well, but there are still a few cases left untested. A closer look at the coverage results can show us which parts of the story diagrams were not executed. This mostly comes down to internal story diagram exception handling. But there are still a few test cases that could be added to increase the test coverage. In the Pull Up Method refactoring for instance we can add a test that includes the pulling up of a method that has parameters. This will increase the coverage of the copyOperation story diagram, since now it never entered the “For-Each”-activity that was meant to copy the parameters of the operation.

In the Extract Class refactoring we can increase coverage by adding an access expression, that is linked to one of the fields that will be extracted and has a child expression. (e.g. something like: `int i = field + 4;`)

However we conclude that the most important cases are tested and that they are successfully performed. From this we conclude that the story diagrams are equivalent to the implementation of the refactorings in eclipse, at least in those cases that were tested.

Figure 6.8: Screenshot of the test results. Shows the number of JUnit tests and the line coverage.
7.1 Summary

In this thesis we have studied how refactorings that are implemented in eclipse can be formalized using the story diagram notation. We have extracted the preconditions and the mechanics for three refactorings implemented in eclipse (Pull Up Method, Encapsulate Field and Extract Class). This extraction is done by inspecting the source code and testing the refactoring itself on some small sample code. The preconditions and mechanics that were extracted have then been formalised in the form of story diagrams. Using Fujaba these story diagrams are executable and we can therefore verify that they behave in the same way as their counterparts in the eclipse implementation.

7.2 Conclusions

The architecture of the eclipse refactoring framework makes it straightforward to extract the preconditions and mechanics of the implemented refactorings. Since all of the refactorings are implemented in the same way, the implementations of the precondition checks and the mechanics can be found in the same places in each refactoring. Once we have the preconditions and mechanics of a refactoring they can be written in a formal notation to create the formal
7.3. THREATS TO VALIDITY

Refactorings are transformations of programming code. And since programming code can easily be translated into a graph representation, it seems like a good idea to represent refactorings as graph transformations. Therefore the formal language that we chose to specify our refactorings in are story diagrams. This is an intuitive (visual) notation, based on graph transformations, that is relatively easy to understand and therefore a good formalism to formally specify refactorings.

We have created the story diagrams for three representative refactorings and have subsequently validated that these story diagrams have the same effect on java program graphs as eclipse’s refactorings have on java code. Thus we have shown that it is feasible to formalise the refactorings that have been implemented in eclipse.

However a few issues remain. The meta-model that we used is an incomplete meta-model for the java programming language. As such not every precondition that we found could be formalised. For instance we were unable to formalise the precondition “Source is not an enumeration.” of the Pull Up Method refactoring as we have no way of representing enumerations in this meta-model.

Another issue is that the story diagrams get parameters as input, whereas the refactorings in eclipse work on whatever is selected in the workspace. This difference in input mechanisms has an influence when the refactorings in eclipse work on a large selection. Take for instance the Extract Method refactoring, this works on a selection of statements and expressions. To give this as input to the story diagrams, the entire subgraph that represent these statements and expressions need to be passed as a parameter. Subsequently we have to add preconditions in the story diagram to verify that these statements and expressions are ordered the right way.

7.3 Threats to validity

Story diagrams can be seen as a visual programming language. As such all issues of programming with a normal language also occur here. This means that the story diagram never works from the first time. It needs to be tested and debugged in several iterations. And even then it can never be guaranteed that the story diagrams are bugfree.
Additionally we can only say that the story diagrams react in the same way as the implementation in eclipse in the cases that we tested. There are still cases left untested that may produce different results.

7.4 Future Work

This thesis paves the road for a few interesting research questions.

How can we assure that the story diagrams preserve the well formedness and consistency of a java program graph? In other words is the java program graph after the refactoring still a representation of a valid java program?

Can we use the story diagrams for primitive refactorings to create more complex refactorings? For instance the Extract Class refactoring is merely a combination of a number of Move Field refactorings followed by an Encapsulate Field for each of the moved fields.

Can we use the implementation of refactorings in eclipse for a formal specification of language independent refactorings?

Another thing that can be done is a more thorough comparative study on the different ways we can formalise refactorings. For instance one experiment that can be done is on the understandability of formal specifications. We take a group of relatively well trained people (e.g. Master students in computer science) and give them a formal specification of a refactoring in the algebraic way and a formal specification of a refactoring in the form of story diagrams. The speed and the accuracy with which the preconditions and postconditions are identified in the formal specification is then an indication of the understandability of the formal specification. This experiment could be used to validate our claim that a formal specification method based on graph transformations is better understandable than the other ways of formalising refactorings.

Having shown that it is possible to formalise the refactorings that are implemented in eclipse, we can use this methodology to formalise all eclipse’s refactorings. This way we can create a catalog that formally describes the most commonly used java refactorings, which can serve as a standard reference when working with java refactorings.


[16] Leif Geiger and Albert Zündorf. Graph transformation-based refactorings using fujaba. 4th International Workshop on Graph-Based Tools (GraBaTs), 2008.


Appendices
Meta-model For Java Program Graphs

Figure A.1 shows the meta-model that we have chosen to represent java code as a graph. This meta-model was an extention by Javier Pérez[22] of Mens’ meta-model[19]. Pérez added language specific elements for java programs such as visibility, packages, interfaces, ... This meta-model also adds a more detailed representation of method bodies. In this thesis we have slightly adapted this meta-model in order to represent nested classes in the Pull Up Method refactoring. This was done by adding a “belongsTo” link between two Classifiers with multiplicity many-to-one since a class can only be nested in one class, but a class can declare many nested classes.

Concepts that are not represented in this include:

- Anonymous Classes
- Exception handling
- Generic Types
- Annotations
- Enumerations
- Type casting
- loops and branches

Constructors are represented as normal methods with a special naming convention (i.e. the method name for a constructor is the same as the name of the class for which it is a constructor).
Figure A.1: Meta-model for the Java Program Graphs.
B.1 Helper Diagrams for the Pull Up Method Refactoring

B.1.1 Signature

This story diagram creates a string form of the signature of a method. Take for instance the following method:

```java
public int someMethod(int i, double d){
    //do some stuff
}
```

If we use this story diagram to find the signature of this method we would get: “someMethod(int,double,)”
Figure B.1: Story pattern that returns a String with the signature of an operation.

B.1.2 checkAccesses

The story diagram (figure B.2) that checks if the variable or method that is accessed or called by a given expression is accessible from the class named container. The given expression can either be an Update expression, an Access expression or a Call expression. Any other expressions are ignored by returning true. If the expression was an Update or an Access expression, then the variable they are linked to needs to be an attribute (i.e. \_this == true). If the variable is not an attribute, then the expression is ignored, since it accesses a local variable and this will be pulled up along with the rest of the method body. If the variable was indeed an attribute, then it needs to be declared in container or in a superclass thereof.

We do a similar operation for checking the method calls. If the method is declared in an interface, then container or one of its superclasses needs to implement that interface. If the method is declared in a class, then that class needs to be container or one of its superclasses.
Figure B.2: The story diagram that checks if fields and methods accessed in a given expression are accessible from a given class.

B.1.3 checkTypes

The story diagram (figure B.3) that checks if the type of a given variable is accessible from a given class. If the type of the variable is nested, we do a similar operation as in the checkAccesses story diagram, namely we check that the type is declared in container or one of its superclasses.

If the type was not nested, then there is only one case in which it might not be visible to container and that is when the type belongs to a different package and is declared package visible (i.e. visibility default).
Figure B.3: The story diagram that checks if the type of a given variable is accessible from a given class.

### B.1.4 copyOperation

This story diagram (figure B.4) creates a copy of the given operation in the given target class.
Figure B.4: Story pattern that copies an operation in a class.
B.1.5 destroy

This story diagram (figure B.5) recursively destroys an expression and all its subexpressions. If the expression was a call it destroys the actual parameters and if the expression was an access to a literal the literal will also be destroyed.

Figure B.5: Story pattern to destroy the expressions in a method.
B.2 Helper Diagrams for the Encapsulate Field Refactoring

B.2.1 checkValidJavaIdentifier

This story diagram checks if a given string is a valid Java identifier. In Java, all identifiers must begin with a letter, an underscore, or a Unicode currency character. Any other symbol, such as a number, is not valid. Furthermore, an identifier cannot have the same spelling as one of Java’s reserved words. In this story diagram we use the static method `Character.isJavaIdentifierStart(char ch)` to determine if the first character of the string can be used as the first letter of a Java identifier. We also verify that the string is not the empty string and that it does not equal any of Java’s reserved keywords\(^1\).

\(^1\)The Java keywords used here were taken from [http://mindprod.com/jgloss/keyword.html](http://mindprod.com/jgloss/keyword.html)

![Story Diagram](image)
B.2.2 createSetter

This story diagram works analogous to the diagram in figure 5.6. First it looks for an existing setter, if it did not find one it will create one. The body of this setter is an update of the variable, by accessing the parameter. Once we have a setter, either a new one or the existing one, we need to convert all updates of the variable (except the update in the setter’s body) to a call to the setter. This is done by a call to the story diagram in figure B.8.

Figure B.7: The story diagram to create a setter in the Encapsulate Field refactoring.
B.2.3  encapsulateSetter

This story diagram is analogous to the diagram in figure 5.7. It effectively changes a given update to a call to the given setter operation. If the `useAccessors` boolean was false, it will verify that the given update expression was not inside the given `container` class.

![Diagram of encapsulateSetter](image)

Figure B.8: The story diagram that changes field updates to setter calls in the Encapsulate Field refactoring.
B.2.4 getAccessorVisibility

This story diagram deals with the visibility aspect of change 2 in the list of mechanics for the Encapsulate Field refactoring (section 4.3.2). The return value of this story diagram is used as the visibility of the getter and setter for the field. If the field was public, then the getter/setter need to be public so this story diagram returns “public”. If the field was private, then the getter/setter may be whatever the user wishes, so it returns accessorVisibility, which was given as a parameter to the perform story diagram. If the visibility of the field was either protected or default, then the visibility of the getter/setter needs to be either the same as the field’s visibility or public. If the requested visibility (accessorVisibility) was neither, then it will just return public.

![Diagram](image)

Figure B.9: The story diagram that returns the visibility for the getters and setters in the Encapsulate Field refactoring.
B.3 Helper Diagrams for the Extract Class Refactoring

B.3.1 getHighestVisibility

This story diagram is used to determine the visibility for newField in the Extract Class refactoring (i.e. changes 6.1 to 6.4 of the mechanics listed in section 4.4.2). It returns the highest visibility of the visibilities of the fields in fields. The ordering is private < default < protected < public. The local variable visibility (of type String) is used to remember what the highest visibility so far is. Initially visibility is set to private (i.e. the lowest visibility). If we encounter a field that is public then we don’t need to set visibility, since public is the highest visibility we can encounter and therefore we just stop our iteration and return public (change 6.1). If we encounter a field that is protected, then we set visibility to protected if in the rest of the iteration no public fields are found, the result of this story diagram will be protected (change 6.2). If we encounter a field that is default and visibility is still private then visibility will be set to default. If in the rest of the iteration no protected or public fields are encountered the result of this story diagram will be default (change 6.3). If in the iteration no public, protected or default fields are encountered, the story diagram will return the initial value for visibility, which was private (change 6.4).
Figure B.10: The story diagram that returns the highest visibility of all fields in the set.
B.3.2 createGettersAndSetters

This story diagram creates a getter and a setter for a given variable and puts them in the given container class. This is analogous to the story diagrams in figures 5.6 and B.7 only now we don’t need to first search for an existing getter and setter. At the end it calls the story diagrams in figures B.12 and B.13. These are analogous to the story diagrams in figures 5.7 and B.8. However we also need to access the new field in order to call the getter/setter.

Figure B.11: The story diagram to create getters and setters for the Extract Class refactoring.
Figure B.12: The story diagram that changes field accesses to getter calls in the Extract Class refactoring.
Figure B.13: The story diagram that changes field updates to setter calls in the Extract Class refactoring.