CODERAIDER
Automatically Improving the Design of Code

NAKUL DURVE
Department of Computing
Imperial College, London

Supervisor: Dr. Emil Lupu
Second Marker: Dr. Naranker Dulay

June 15, 2007
Abstract

Software is subject to continuous modification to comply with changing requirements. Every change made inevitably degrades the quality of the design of software, and so aggressive measures need to be taken to counter this effect and improve the software design. This report outlines an innovative method towards improving the design of code by refactoring, using an approach which enables the automation of three things - finding where code needs to be improved, identifying how it might actually be improved, and actually performing the improvement. This will be of benefit to software maintainers who often have to improve code, but have no approach available which provides automation for all three of these tasks. This approach works at a much higher structural level on classes in the code, a feature which many existing approaches towards code improvement do not encompass. This enables a good degree of support for object oriented design practices, which is crucial to successfully improving the overall design of code.

The technique designed in this project enables developers to write refactorings as code examples, the principles of which can be applied to real projects. This has the advantage that code examples can be extracted from existing projects, or even shared between developers working on similar projects. This extensibility overcomes many challenges related to designing a code improvement system which is flexible enough to be universally applicable. Indeed, the suggested methodology also includes support for variation, so code which is slightly dissimilar to the refactoring code examples can still be analysed and actioned appropriately.

The ideas contributed by this project to the field of automated code improvement are supplemented by a concrete implementation of a tool integrated with Eclipse, which can analyse code, and perform appropriate improvements based on a knowledge base of code examples. We shall analyse the practicality of the suggested approach, and show that it has great potential to be of real benefit to existing projects. We will demonstrate several code improvements which can be easily specified within, and performed by, CoderAider in a manner which not only improves quality, but also highlights the inability of existing techniques to perform the same quality enhancements. A case study will be taken, namely Apache Tomcat, where the code improvement features of CoderAider will become evident.
Acknowledgements

I would like to thank my supervisor, Dr. Emil Lupu, for his invaluable advice and insight on various aspects of the project, for agreeing to supervise this project, and for taking the time to read this report thoroughly and suggest a number of useful improvements. I would also like to thank Dr. Naranker Dulay, my second marker, for providing some useful tips about the direction of the project.

Finally, I would like to thank the developers of various open source tools and frameworks that proved highly useful during the implementation phase, as well as the authors of the articles in the “Eclipse Corner”, which proved to be a useful starting point for learning about developing for Eclipse.
# Contents

Abstract iii  
Acknowledgements v  
Contents vii  

## 1 Introduction
1.1 Historical Perspective ........................................... 1  
1.2 Software Maintenance ........................................... 2  
1.3 Methods of Reducing Software Entropy ........................ 3  
1.4 Motivation ......................................................... 4  
1.5 Contributions ..................................................... 4  

## 2 Background
2.1 Design Patterns .................................................. 8  
2.1.1 Abstract Factory as an example of a design pattern ........ 8  
2.1.2 Characteristics of a good design pattern .................... 9  
2.1.3 Criticisms of design patterns ................................ 10  
2.1.4 Domain Specific design patterns ............................. 10  
2.2 Refactoring ....................................................... 11  
2.2.1 Examples of refactoring ...................................... 11  
2.2.2 Problems with refactoring .................................... 11  
2.3 Detecting Refactoring Requirements ............................ 12  
2.3.1 Bad Smells ..................................................... 13  
2.3.2 Using Version Control ........................................ 13  
2.3.3 Using Constraints on Design ................................. 14  
2.3.4 Static Analysis ............................................... 15  
2.3.5 Search Based Algorithms .................................... 19  
2.3.6 Metrics ......................................................... 20  
2.4 Currently Available Tools and Frameworks .................... 22  
2.4.1 ArgoUML ....................................................... 22  
2.4.2 PMD .......................................................... 23  
2.4.3 FindBugs ....................................................... 24  
2.4.4 Jackpot ......................................................... 24  
2.4.5 Parsers and Parser Generators ............................... 25  
2.4.6 Transmogrify ................................................ 29
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.7</td>
<td>OpenJava</td>
<td>30</td>
</tr>
<tr>
<td>2.4.8</td>
<td>JavaXL</td>
<td>30</td>
</tr>
<tr>
<td>2.5</td>
<td>Refactoring Techniques</td>
<td>31</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Program Slicing</td>
<td>31</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Graph Transformations</td>
<td>32</td>
</tr>
<tr>
<td>2.5.3</td>
<td>Strategy Based Elimination</td>
<td>32</td>
</tr>
<tr>
<td>2.5.4</td>
<td>Composition and Dependencies between Refactorings</td>
<td>33</td>
</tr>
<tr>
<td>2.5.5</td>
<td>Behaviour Preservation</td>
<td>34</td>
</tr>
<tr>
<td>2.6</td>
<td>Language Independent Refactoring</td>
<td>35</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Meta-modelling</td>
<td>35</td>
</tr>
<tr>
<td>2.7</td>
<td>Summary of Background</td>
<td>37</td>
</tr>
<tr>
<td>3</td>
<td>Overall Design</td>
<td>39</td>
</tr>
<tr>
<td>3.1</td>
<td>Aims</td>
<td>39</td>
</tr>
<tr>
<td>3.2</td>
<td>Interaction</td>
<td>41</td>
</tr>
<tr>
<td>3.2.1</td>
<td>The lifecycle of a refactoring</td>
<td>43</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Reasons for supporting Java</td>
<td>44</td>
</tr>
<tr>
<td>3.3</td>
<td>Performing the Refactoring</td>
<td>44</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Step 0 - Refactoring Specifications</td>
<td>44</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Step 1 - Matching</td>
<td>45</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Steps 2 &amp; 3 - Transforming the source code</td>
<td>46</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Step 4 - Rewriting the source code</td>
<td>46</td>
</tr>
<tr>
<td>3.4</td>
<td>Summary</td>
<td>47</td>
</tr>
<tr>
<td>4</td>
<td>Eclipse Infrastructure</td>
<td>49</td>
</tr>
<tr>
<td>4.1</td>
<td>Plugin Framework</td>
<td>49</td>
</tr>
<tr>
<td>4.2</td>
<td>Eclipse Java Model</td>
<td>50</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Making modifications to the Eclipse Java Model</td>
<td>51</td>
</tr>
<tr>
<td>4.3</td>
<td>Refactoring Framework</td>
<td>52</td>
</tr>
<tr>
<td>4.3.1</td>
<td>CoderAider Components</td>
<td>53</td>
</tr>
<tr>
<td>4.4</td>
<td>CoderAider integration into Eclipse</td>
<td>53</td>
</tr>
<tr>
<td>4.5</td>
<td>Summary</td>
<td>57</td>
</tr>
<tr>
<td>5</td>
<td>Refactoring Specification</td>
<td>59</td>
</tr>
<tr>
<td>5.1</td>
<td>Desired expressibility</td>
<td>59</td>
</tr>
<tr>
<td>5.2</td>
<td>The XML Approach</td>
<td>60</td>
</tr>
<tr>
<td>5.3</td>
<td>The Templates Approach</td>
<td>61</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Template Annotations</td>
<td>61</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Improvement Templates</td>
<td>63</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Design Pattern Templates</td>
<td>64</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Templates as Critics</td>
<td>64</td>
</tr>
<tr>
<td>5.3.5</td>
<td>Template Processing</td>
<td>64</td>
</tr>
<tr>
<td>5.4</td>
<td>Summary</td>
<td>66</td>
</tr>
</tbody>
</table>
## 6 Model Design

6.1 Design Issues ................................................................................. 67
   6.1.1 Custom Model or Existing Model? ........................................... 67
   6.1.2 General Issues ......................................................................... 68
6.2 Generic Model .............................................................................. 68
   6.2.1 Type Descriptors ..................................................................... 68
6.3 Java Specific Model ....................................................................... 70
6.4 Model Generation ......................................................................... 71
6.5 Model of a template ...................................................................... 72
6.6 Summary ......................................................................................... 72

## 7 Matching of Patterns within Code

7.1 Classes .......................................................................................... 74
   7.1.1 Super classes ........................................................................... 75
   7.1.2 Attributes ................................................................................ 75
   7.1.3 Methods .................................................................................. 77
7.2 Dependency Resolution and 2-phase matching ................................... 78
7.3 Properties of the matching process .................................................. 80
7.4 Result of the matching process ......................................................... 80
7.5 Summary ......................................................................................... 81

## 8 Code Transformations

8.1 Transformation of Classes ............................................................... 84
   8.1.1 Transformation of Attributes .................................................. 84
   8.1.2 Transformation of Methods ..................................................... 85
   8.1.3 Transformation of Superclasses .............................................. 85
8.2 Transformations Wrapper ............................................................... 86
8.3 Summary ......................................................................................... 86

## 9 Metrics Framework

9.1 Architecture .................................................................................. 87
9.2 Defined Metrics ............................................................................. 88
9.3 Extensibility .................................................................................. 89
9.4 Summary ......................................................................................... 90

## 10 Testing

10.1 Strategy ......................................................................................... 91
10.2 Test cases ..................................................................................... 92
   10.2.1 Overridden Attributes ........................................................ 92
   10.2.2 Overridden Method calls in constructors ............................... 93
   10.2.3 Improving a Singleton .......................................................... 93
   10.2.4 Removal of a null final field ................................................ 93
   10.2.5 Type Safety ........................................................................... 96
   10.2.6 Chain of Responsibility ........................................................ 99
10.3 Demonstration on Apache Tomcat .................................................. 101
   10.3.1 Area of analysis ................................................................... 101
   10.3.2 Problem ................................................................................. 101
Chapter 1

Introduction

“If debugging is the process of removing bugs, then programming must be the process of putting them in.”

- Edsger Dijkstra

1.1 Historical Perspective

In 1995 Denver International Airport, in conjunction with BAE Automated Systems, began developing an automated baggage handling system which would autonomously route luggage through several miles of conveyors for delivery to passengers and planes. The system cost hundreds of millions of dollars, and its launch was delayed by over a year. Several faults were identified after the system was put into use, and after years of tweaking, the project was finally scrapped, resulting in a severe financial loss to the airport, as well as bankruptcy for the software consultancy firm responsible. Such is the high price of poor software design. Several other cases can be observed where hundreds of millions of dollars, perhaps even billions, were spent on systems doomed to be scrapped, delayed by several years, or plagued with fault.

Recent examples include the £6.8bn IT upgrade of the NHS which is falling behind schedule, and estimated to cost at least double the original estimate. The Inland Revenue was also the victim of such software errors, and a problem with the tax credits system in 2005 resulted in a loss of £94m. Over in the United States, the U.S. Internal Revenue Service saw a failed $4bn modernization effort earlier last decade, followed by a troubled $8bn updating project.

Failures such as those discussed above may occur due to several reasons, but often central to this is poor planning and allocation of resources. Effort is often underestimated, and features have to be scrapped to meet deadlines. Communication is often inadequate which can lead to a failure in identifying, documenting and tracking requirements. Crucial design flaws are discovered too late, usually as a result of a poor understanding of the requirements of the system, but sometimes also as a result of a late change in system requirements. Inadequacies are often discovered once the system becomes functional. This results in severe delays in launching the system because the critical design elements need to be modified. This generates a huge problem, as the overall design is often poorly thought out, and not meant to be adapted in such a manner so as to incorporate newer requirements, or so as to solve a problem slightly different to the original one. Often, tight deadlines for delivery mean that quality assurance and testing phases are skipped or scaled back. This permits errors in the production environment, which can be problematic later on for product support teams.

This shows that one of the key areas where problems exist in industry is in the development and support of new software. Requirements are continually being updated and added, and the software designers must be able to cater for these through the process of software maintenance. In the mid-1990s, software maintenance was thought to have accounted for over half of the $160bn spent annually on software development, as developers had to spend a vast amount of time to study software systems and
their implementations before deciding how to make the required changes. Therefore, software maintenance should be as trouble free as possible, and the ways in which this is done are subject to heavy debate; as yet, no single concrete solution exists, although several methods have been identified which simplify software maintenance. Tools which reverse engineer the system to produce a higher level of abstraction are quite common, and a good starting point in the maintenance process.

1.2 Software Maintenance

In a real world environment, due to continually changing requirements, existing software is subject to continuous change. Maintenance and enhancement of software consumes a substantial portion of the total life cycle cost of a system - some analyses have estimated up to 80%. Swanson [37] defined a taxonomy for the major types of software maintenance.

- **Perfective maintenance** - to perfect the system in terms of its performance, efficiency or maintainability.
- **Adaptive maintenance** - to adapt the system to changes in its data or processing environment.
- **Corrective maintenance** - to correct performance, implementation or processing failures of the system.

It should be clear that the categories of maintenance identified here are mutually exclusive, but in a real world scenario, a single maintenance job will often fall under several of these categories as many tasks are undertaken at once. The reason for undertaking several tasks at once is simple - the programmer will be familiar with that region of code, and better positioned to make any required changes there, whatever category of maintenance they fall under. Often, new tasks will be identified for programmers to work on when they are in contact with a certain region of code, as allocating resources for those new tasks in the future is more expensive, as programmers have to re-familiarise themselves with that region of code.

During maintenance, developers are often expected to implement solutions for systems which they did not design, and have poor documentation. This means they must spend time analysing the system, and any inter-dependent relationships between the individual system components. Tight schedules often mean that any code they write to solve the problem at hand is ad hoc, and only a temporary solution. Over time, more and more ad hoc patches are applied to the system. It is for this reason that, as a result of maintenance, the original design inevitably degrades, and the codebase becomes more and more complex. This has been referred to as the software entropy effect [4], drawing a parallelism from the Second Law of Thermodynamics where the entropy of a system increases with time. This is one of the prime reasons why so much effort is required for software maintenance.

Lehman [22] captured these properties of software evolution with five laws:

1. **The law of continuing change** - In order for a system to maintain its usefulness, it must change continually.
2. **The law of increasing complexity** - Modifications result in an increase in complexity, leading to an increase in resource allocation to manage the increased complexity.
3. **The law of large program evolution** - Program evolution is a self-regulating process, and factors such as size, time between releases, errors are invariants per release of the system.
4. **The law of organizational stability** - During the lifecycle of a program, its rate of development is constant and independent of the resources devoted to it.
5. **The law of conservation and familiarity** - Inter-release changes are incremental and do not fundamentally impact maintainability.
1.3. Methods of Reducing Software Entropy

From this, we can see that care must be taken to ensure that any patches applied to the system do not degrade the original design; this becomes increasingly complicated over time, given shifts in programming paradigms, and an incomplete awareness of the full system functionality. Thus, over time, as more requirements become evident, the complexity of the software increases, making it harder to maintain. Therefore we observe a vicious upwards spiral in complexity - requirements increase, leading to more maintenance work. As each iteration of maintenance work degrades the design and adds complexity, future maintenance work becomes harder. To cope with this spiralling complexity, there is an urgent need for techniques that reduce software complexity by incrementally improving the internal software structure [24], i.e. reducing software entropy.

1.3 Methods of Reducing Software Entropy

Rewriting is a common method of reducing software entropy. It involves scrapping the existing codebase and redesigning it, with the added advantage of hindsight. In theory, this may seem better, and while this can have the required benefits of reducing entropy and making the code easier to maintain, it is very time consuming, and requires thorough regression testing to be performed as both the functionality and the structure of the code are likely to have changed, meaning new problems may have been introduced. This increases maintenance costs substantially, especially if it has to be done at regular intervals. Thus, a better approach is required which has the same entropy reducing effect, but fewer overheads and side effects. This better approach comes in the form of refactoring.

Refactoring is a powerful technique which minimizes software entropy by use of a disciplined approach to restructuring code. Restructuring is the transformation from one form to another while preserving the system’s external behaviour in terms of functionality and semantics. A restructuring transformation is often one of appearance such as altering code to improve its structure in the conventional sense of structured design [6]. It may be thought of as a form of preventative maintenance, meant to decrease the complexity of the system in anticipation of Lehman’s first law. Refactorings are also local transformations, and do not affect the developer’s familiarity with the system, in accordance with Lehman’s fifth law.

Refactoring was first studied by Opdyke [29]. It neither fixes bugs nor adds new functionality, rather it improves the readability of the code by changing its design and structure. Fowler [14] describes it as a controlled technique for improving the design of an existing codebase by applying a series of small behaviour preserving transformations, each of which are too small to be worth doing, but the cumulative effect of which can be quite significant. Indeed, it is an integral part of widely used modern day programming techniques such as extreme programming and other agile methodologies, where developers are encouraged to alternate between adding functionality, and refactoring the code, to improve its internal clarity and consistency. By continually refactoring a codebase, software maintainability becomes easier because the design stays flexible and becomes streamlined due to the changes in its internal structure. Refactoring can be thought of as a process which counteracts software entropy by overcoming unnecessary complexity, architectural limitations, changing requirements and a flawed or incomplete understanding of the system. Logic is often simplified during the process of refactoring, and newer functionality can become easier to add as a result of restructuring.

Unfortunately, refactoring is a time consuming activity. Developers have to go through the code and restructure areas carefully, in a way that adds some benefit in terms of maintainability and flexibility, but also preserves behaviour. Thorough testing is needed after each refactoring, and automated unit tests can be useful. This overhead is the major downside of refactoring, and is often the reason why companies are reluctant to devote resources to it.

Automated refactoring tools exist which perform a refactoring on some code, provided the programmer selects the area of code and the refactoring to apply. This means human interaction is required. Several automated tools ask the user for input whilst doing the refactoring - making the wrong decisions at this stage can introduce errors into the code. It is this human intervention which often means that programmers are reluctant to use these tools. While automated refactoring tools consume less development time than manual refactoring, they do still require thought on the part of the programmer, and this inevitably means that resources have to be allocated to this activity of system refactoring.
1.4 Motivation

This section has briefly highlighted the importance of software maintenance to any development project, and has shown examples where lack of appropriate maintenance and development techniques has resulted in the catastrophic failure of large projects. Refactoring has been introduced as a means of simplifying the software maintenance aspect. This project seeks to further explore refactoring, and its benefits to software maintenance, and the feasibility of developing a fully automated refactoring tool which can both suggest and implement refactorings on a given codebase in order to improve its design and/or reduce the likelihood of bugs within the code. Such a tool could substantially help in the development and maintenance of code, as seen previously, without the requirement for the additional overheads of the existing refactoring tools.

In particular, this project will attempt to improve code by making use of existing knowledge about object-oriented development. This is enterprising because there are potentially a vast variety of development ideologies and desired code properties related to different domains, and this project needs to create a way to utilise (both capture, and use) this existing knowledge in order to perform improvements. This project will also attempt to encourage and automate the sharing of good practices between developers - for example, problems and solutions common to web applications (or found in a particular web application) should be shared and reused on other web applications. This project will try and make it simple to extract elements of good design from one application, and apply the same principles to other projects. This needs to take care of the difficulties associated with extracting relevant parts of the design, and also making the process very simple for developers to do. Most importantly, this project will attempt to integrate any methodology with existing software developing practices, in order to encourage its widespread use. This is challenging because it needs to be universally applicable, and easily understood by anyone with a programming background.

1.5 Contributions

This project contributes an innovative yet simple methodology for the representation of good design practices. A unique way of allowing developers to provide examples of desired coding practices, or examples of how to change existing coding practices, is demonstrated, and a method is devised to apply the principles of these examples onto real projects. What distinguishes this method from existing tools is the simplicity involved in specifying these examples, and the ability for existing projects to serve as the examples for either good or bad practices. In this way, the novel idea contributed by this project has the major advantage that the design ideas used in existing projects can serve as learning material for analysing newer projects. This also makes it simple for domain specific knowledge to be shared between developers of that domain. One of the major hurdles encountered and overcome was that of extensibility - the ideas contributed by this project allow newer coding practices, or domain specific practices, to be added and utilised with minimal effort. Many of the practices illustrated are unique to this project, and other tools cannot encompass such behaviour. Full details of the mechanism behind this specification methodology based on case studies, are given in Chapter 5.

This project develops a model for the representation of code, which abstracts away the complexity of existing rewriting frameworks, and also makes it simpler for developers to extend this project to incorporate any necessary functionality, thus surmounting the existing problems with over complicated code models and transformation frameworks which require an indepth understanding of low level implementation structures. The example of code metrics is taken, where numerical analysis is carried out on the code to gauge areas for improvement, or analyse the code quality. A flexible and extensible approach towards code metrics definition will also be demonstrated, highlighting the power of this model for multipurpose use, the ease with which new metrics may be added, and showing how code metrics may be used in conjunction with the novel ideas of good practice specification in order to further improve the way in which code can be analysed and improved. Such an approach may involve heuristical analysis in order to generate a likelihood quality estimate for the code, and potential regions for improvement. This contribution of the project deals with the problems associated with encountering a wide variety and
1.5. Contributions

style of possible code metrics, by extracting common functionality, and providing the capability to easily add and extend existing functionality for code metrics analysis, which may be incorporated into the code analysis stage with ease. More details are given in Chapters 6 and 9.

To demonstrate the practicality of the suggested ideas, this project contributes a plugin for the Eclipse environment that has a multitude of functionality. Firstly, it is capable of analysing source code to determine potential regions for improvement and find potentially flawed areas, at the structural level of the code. Chapter 7 outlines the mechanism behind this. Secondly, it can suggest how to proceed with improving these areas, as shown by Chapter 8. Thirdly, the plugin contributed by this project integrates into the existing Eclipse refactoring framework, and is capable of showing the user a preview of the proposed changes, and integrates with the history of performed refactorings stored in Eclipse, enabling for easy reversal of changes. Chapter 4 demonstrates this functionality. This third contribution largely overcomes the difficulties associated with encouraging widespread use of the ideas behind this tool, as it overcomes the obstacles of usability and developer trust in automated tools, by providing good integration with a commonly used development environment, and ensuring all changes are idempotent and easy to observe and reverse. Lastly, it provides an implementation for one of the other major contributions of this project - namely that of utilising the code model to generate code metrics, which may be used to improve the analysis phase.

Finally, the practicality and scalability of the methodology is analysed, and we see that it has potential for use on real world projects. An example analysis is carried out on the Apache Tomcat source code, where a bug is found and fixed automatically. This is shown in Section 10.3.
Chapter 2

Background

“Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

- Brian Kernigan

In the previous chapter, the importance of software maintenance to industry was considered, and refactoring was highlighted as one of they key ways of making the maintenance process easier. This chapter shall focus on the design of software and refactoring in more depth, and the various techniques which have been developed to automate refactorings, and aid in the process of refactoring automation. A survey will be conducted of state of the art methods in this field to give the reader an idea of the general progress that has been made in this area, and a better idea of the current problems in this area. The aim of this chapter is to provide a firm basis for developing a specification for this project which can fulfill the aims already outlined, based on where the major issues lie in the field.

This chapter is divided as follows - the first part of this chapter deals with well established tools and theories about design, and the rest of this chapter deals with the research currently being conducted, and different state of the art techniques. The relevance to this project is considered at each stage.

- **Design Patterns** - Section 2.1. This section explores the concept of a design pattern, and what it means for a design to be considered as good or bad. Examples are given, and design patterns specific to certain domains are also considered.

- **Refactoring** - Section 2.2. This section explores the different kinds of refactorings available, and considers the pitfalls of refactoring.

- **Detecting Refactoring Opportunities** - Section 2.3. This section considers the ways in which it is possible to automatically detect that a certain part of the codebase needs refactoring. In other words, methods of detecting poor or inefficient design are considered.

- **Tools and Frameworks** - Section 2.4. This section look at the existing tools and frameworks which help in the process of refactoring - from the point of view of the developer, and from the point of view of a refactoring tool designer. This includes modern IDEs which offer some automation of refactorings.

- **Refactoring Techniques** - Section 2.5. This section looks at the ways in which refactorings can be applied automatically, once the offending region of code has been identified. Methods of identifying the best refactoring for the given region are also considered, along with methods of applying more than one refactoring to the region of code.

- **Language Independence** - Section 2.6. This section looks at the feasibility of developing a language independent solution to automated refactoring.
2.1 Design Patterns

The idea of applying a design pattern to software was first considered by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides \(^1\) [15]. They describe a design pattern as a “proven object oriented solution to a frequently occurring problem, which identifies and specifies abstractions above the level of single classes and instances”. Gamma et al. categorised patterns into three main areas:

- **Creation**al - these create objects for the developer instead of having to instantiate them directly, resulting in greater flexibility in choosing objects required for a given scenario.
- **Structural** - these help compose groups of objects into larger structures.
- **Behavioural** - these help define the communication between objects in the system.

Design patterns reflect the insight and knowledge of developers who have already successfully used these patterns in their own work to solve a problem. The solution provided is a ready-made reusable one, capable of being useful in different situations. They are also quite expressive in that they provide a common repository of solutions which expresses larger solutions succinctly. Usage of design patterns often indicates good design, as that design has been proven to be successful (either in terms of functionality or maintainability). Note that the presence of design patterns is not guaranteed to mean good design, but is only a likely indicator. Similarly, the absence of design patterns is not an indicator of a poor design, but may be a likely indicator that the design has not been thought out properly. It may however be the case that the design works very well even without the presence of design patterns, and this is because there are several varieties of design patterns, some suited to specific contexts, and these are looked at later.

2.1.1 Abstract Factory as an example of a design pattern

An example of a design pattern is the abstract factory pattern, as shown in Figure 2.1. This provides an interface for creating families of related or dependent objects without specifying their concrete classes [9]. In conjunction with Figure 2.1, consider the following example, taken informally from [9].

- **AbstractFactory** (e.g. ContinentFactory) - declares an interface for operations that create abstract products.
- **ConcreteFactory** (e.g. AfricaFactory, AsiaFactory) - implements the operations to create concrete product objects.
- **AbstractProduct** (e.g. Herbivore, Carnivore, Omnivore) - declares an interface for a type of product object
- **Product** (e.g. Lion, Deer, Wolf, Tiger) - defines a product to be created by the corresponding concrete factory, and implements the AbstractProduct interface
- **Client** (e.g. AnimalWorld) - uses interfaces declared by AbstractFactory and AbstractProduct classes

From this example, it should be clear that an abstract factory provides a means of encapsulating groups of factories that have a common theme. The client is expected to create concrete implementations of the abstract factory, and use generic interfaces to create concrete interfaces part of that theme. The client should not care which concrete objects it gets from the internal factories as it uses a generic interface to them. This results in the separation of implementation details from usage details [42].

\(^1\)These authors are often referred to as the Gang of Four, or the GoF
2.1.2 Characteristics of a good design pattern

A design pattern is thought of as being good \([7]\) if it:

- Solves a problem - the pattern should capture a solution instead of an abstract principle or strategy.
- Is a proven concept - the pattern should have a track record of having solved a particular problem - theories and speculation are not allowed.
- Is not obvious - the best patterns attempt to generate a solution to a problem indirectly, possibly by deriving from first principles.
- Describes a relationship - patterns describe systems as a whole, including their structure and inter-relationships between objects.
- Has a significant human component - patterns should make the life of the developer easier by appealing to utility and aesthetics, often with the purpose of making maintenance easier.

Design patterns exist at all levels - from very low level specific solutions to broadly generalized issues. Many are applicable across a wide spectrum, but some solve just a single problem. This spectrum is considered further in the Section 2.1.4. It should be noted that the use of patterns does not guarantee success. A pattern may often need to be adapted to best fit the problem at hand. Experience and understanding of the pattern and the problem are crucial when deciding how the pattern can improve the design. An interesting subset of design patterns are antipatterns - these are similar to design patterns, but differ in the sense that they describe known poor solutions to solving a problem. Examples include the modifying of parameters passed into a method, and superclasses referring to their subclasses.

It should be obvious that any system capable of suggesting and implementing refactorings automatically should have the ability to draw on this knowledgebase of design patterns, as they are concepts proven by human experience. Expecting the refactoring to derive these patterns is out of the question, as it would involve the system to have to analyse the problem, probably from a semantic perspective. If however the system has a knowledgebase of design patterns present, it is possible the system can use this knowledge, and perhaps find variants of patterns, or derive new patterns based on existing ones.
2.1.3 Criticisms of design patterns

Although design patterns have often been looked at as the holy grail of software design, there are a few noteworthy criticisms [45]:

- Target the wrong problem - design patterns often provide too much of an abstraction, or encourage the over-engineering of a problem by adding flexibility which is not required. This may be beneficial in the future for maintenance work, although many agile methodologies condone this and encourage you to develop the quickest thing which could possibly work.

- Lack formal foundations - design patterns have been discussed in the literature in an ad hoc fashion, and the whole area is a very subjective one.

- Lack of reusability - each pattern must be reprogrammed into each application that uses it. This can be viewed as a lack of reusability, as patterns have to be modified slightly and reprogrammed for the application - there is no component or module which can be used generically.

2.1.4 Domain Specific design patterns

Many design patterns found in the wild do not conform to traditional ones described in the literature, but are new variants which have evolved to support domain specific requirements. Two main distinctions can be made when referring to domain specific design patterns:

- Business related - Some design patterns are suited to solving just one problem, while others are suitable in a specific area such as financial applications. Like all design patterns, these arise as a result of experience and hindsight gained from working with the problems at hand.

- Language related - Some design patterns are specific to a certain language or technology. For example, developers of XSLT 2 would cite design patterns based on things like Wendel Piez’s method of non-recursive looping, and Oliver Becker’s method of conditional selection of a value in a single expression.

Due to this, it should be clear that any solution which will help programmers improve their designs should not assume a specific set of design patterns, rather it should be able to cater for new and custom patterns as specified by the programmers.

As an example of a domain specific design pattern, consider the requirements of a financial institution. There are a vast array of complex products such as futures, options, warrants, structured credit products, fund derivatives etc. Many of these products are composed of one another, and often have additional business logic tied in. A traditional design pattern may represent each product separately, extended from a common AbstractProduct. However this may not necessarily be the best solution as the requirements here are wildly different to those where abstractions are normally used. Restrictions on speed of access, caching of data over a network and so on may also impose limitations on the styles of patterns used. As a result, a new architecture may be used here which best represents both the interests of the firm’s business logic, and the software engineers viewpoints on maintainability.

From this, it is clear that any system which has the ability to do automatic refactoring should also have the ability for users to specify any domain specific design patterns which are in use in their projects. This becomes harder when considering language related patterns, as many languages have completely different syntax and are used for different tasks. However, it should be possible for business related patterns as they will mostly be specified in the traditional object oriented sense.

---

2 Extensible Stylesheet Language Transformations (XSLT) is a language for transforming XML documents into other XML documents
2.2 Refactoring

Refactoring was introduced in Section 1.3 as a means of reducing software entropy in the maintenance process. It is basically a “disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour” [13].

2.2.1 Examples of refactoring

Let us consider a few examples of the refactoring process. These are by no means a comprehensive list, but are included to give the reader a flavour of what is meant by refactoring.

- **Pull Up Method** - This “pulls” up a method from a subclass into a superclass. This may be useful when the method needs to be used by several subclasses.

- **Push Down Method** - This “pushes” down a method from a superclass into a subclass. This may be useful when only a single subclass uses that functionality, meaning there is no point making that functionality available to all subclasses. It may even be the case that the functionality in question should only be available in a specific subclass.

- **Move Method** - This moves a method from one class to another. All references to this method are updated accordingly. This can be useful when redistributing functionality, especially when removing antipatterns such as the God Class 3.

- **Extract Method** - This takes a piece of code within a method and places it in a newly created method. This has the advantage of making that piece of code available to clients without “copy and pasting” code. It can also help improve readability as the semantics of the code become clearer. Consider Listings 2.1 and 2.2.

- **Encapsulate Field** - This takes a public field in a class, and reduces its visibility to private, and provides accessor methods to read and modify the field. This is to conform to object oriented principles.

Several similar refactorings can be defined, and new ones emerge all the time as developers realize new types of changes which can be made to the codebase to improve maintainability, or increase efficiency. Detecting when to perform refactorings is described further in Section 2.3.

Any system which has the ability to perform refactorings should be aware of what kinds of refactorings can actually be performed. It need not be aware of a vast number of refactorings, as many can be derived by combining some elementary refactorings together. Thus, the system should have a knowledgebase of available refactorings such as adding or removing classes, methods and attributes, or moving methods and so on. From this, more complicated refactorings could either be derived, or specified by a designer.

2.2.2 Problems with refactoring

Refactoring cannot solve all problems that a developer may have. It is still unclear whether certain design decisions are so central that refactoring cannot help at all. The question then becomes one of changing

---

3 The God class is an antipattern which defines a class that does everything - i.e. it has too much functionality.
the design entirely from one design to another, and whether this is possible with the aid of refactoring. While some may argue it is, time constraints should be taken into account as well. Refactoring itself is a tedious process, and needs thorough testing after it has been performed. It may well be quicker to develop the design from scratch than spend time updating the existing design.

One of the key themes in object oriented programming is the separating of interfaces from implementation. The idea is that implementations of an interface can vary wildly, but as they conform to a certain interface, they will be usable by any client. Certain kinds of refactorings change the interfaces themselves, and this means that clients using that interface have to be updated. Often in large projects, the code for these clients is not available as it is split up between multiple teams and business units, and requires several layers of management to sign off and approve. This is a scenario where refactoring should be avoided. Similarly, it is often the case in businesses that implementation details are so tightly dependent on the design that it is not possible to refactor without causing a break in functionality.

Time schedules should also be taken into account when refactoring. Fowler [14] points out that refactoring when close to a deadline is a bad idea, as the benefits of refactoring would only be present after the deadline has passed, and thus be too late.

2.3 Detecting Refactoring Requirements

When about to embark upon a refactoring quest, the first thing that programmers must do is the identification of where refactoring is required within the program. The same is true for any system which automatically performs refactorings - it must detect which areas of the system require refactoring. This involves detecting possible problems, flaws, or inefficiencies with the code or its design. Much of this is down to intuition, but efforts have been made to formalize this. It is these formal efforts which will be of use in this project, in order to develop a system capable of replicating human refactoring efforts. Being able to automatically detect and correct design flaws in code is a major research area, and the applications for such a system would be large. Refactoring helps reduce the cost of maintenance and makes it easier to make modifications in response to requirements. There is a lack of tools for automating this task of finding design defects.

The first step towards detecting defects automatically is to classify the defects. Yann-Gaël Guéhéneuc et al. [16] classify design defects into 3 categories:

- Intra-class - These are defects within the structure of a class, or its members. They are mainly concerned with the style of coding.

- Inter-class - These are defects related to the structure of the classes and their relationships.

- Behavioural - These are defects related to the semantics of the code.

These classifications can intersect with one another to form new classes of design defects, to better classify those defects which don’t quite fit into one category, or fit into multiple categories. This partitioning
2.3. Detecting Refactoring Requirements

Listing 2.3: Bad smell in code - code duplication

```java
a = a * b - 50;
b = a - b--; // and later on
c = c * b - 50;
b = c - b--; 
```

Listing 2.4: Bad smell in code - dead code

```java
int a;
a = 5; // This line is dead code
a = 6;
b = a * 2;
```

serves to be able to categorise the design defects more accurately, and so make comparison of correction tools easier. However, more kinds of defects need to be considered in this classification to ensure that it can cater for the restructurings generally required.

2.3.1 Bad Smells

Beck and Fowler [14] develop one of the most widespread ways of detecting refactoring opportunities - the concept of “bad smells”. These are “structures in the code that suggest the possibility of refactoring”. In other words, they are regions of code which seem like they may contain some problems. Such smells often contribute to reducing the maintainability of code. Examples include code duplication, dead code, large methods and classes, feature envy, long parameter lists, lazy classes and many more. Code smells can be used to assess the quality of the code, and identify places that need improvement.

Consider the smell of duplicate code. This may occur when code has been copied and pasted, and possible changed slightly. Listing 2.3 demonstrates this. This would be considered a bad smell, as the code is duplicated, and needs to be extracted out into a method (as shown in Listing ??). There is obviously some logic behind that calculation, and this is liable to change. If and when it does change, the programmer should only have to change it in one area. Therefore, this code should be placed into a method that takes one variable as a parameter. Consider also Listing 2.4. There is a line which does not contribute anything to the behaviour of the program, and so should be removed.

The examples given here for code smells are quite easy to spot programmatically, as the concept of what to look for is very clear. It is for this reason that some of these bad smells will be a useful starting point in developing any system which detects refactoring requirements automatically. Unfortunately, not all of Fowler’s code smells can be spotted as easily. Consider the smell of Inappropriate Intimacy. As this is a highly subjective smell, it becomes difficult to detect it automatically in a correct manner.

2.3.2 Using Version Control

Ratzinger et al. [32] claim that bad smells are too subjective, and propose a more concrete method, which extracts historical data from source control repositories and identifies which parts of the system change together, thus indicating high coupling. These parts of the system are then said to be candidates for refactoring. Ratzinger extends Fowler’s bad smells with “change smells”. These are smells not visible in the code itself, but visible when viewing the change history. This occurs when dependencies within the code are not detectable statically, but become obvious during the code evolution as developers change multiple parts of the system together. Ratzinger defines for each pair of objects in the system, a coupling

---

4 Feature Envy - a smell when a class that uses methods of another class excessively
5 Inappropriate Intimacy - a smell when two classes are overly intertwined
metric which counts all log groups in which these objects changed together. For large systems, calculating this for each pair of objects becomes very computationally intensive.

Ratzinger’s method depends on the critical assumption that historical data is available for analysis, and this is often not the case. Even if it is available, this sort of analysis seems open to generating false positives, as developers often commit for other reasons such as fixing comments, or even repository maintenance reasons such as creating / pruning branches. This method does no more than identify where coupling lies in the system. It does not propose any means of identifying if there is a real problem or not, as coupling does not necessarily indicate a problem. As the exact nature of the problem cannot be identified by this method, it will not help us in determining how to go about refactoring code automatically. In Ratzinger’s case study, the refactoring was done manually, once the classes with high coupling were identified, and this requires a significant human component.

2.3.3 Using Constraints on Design

Yann-Gaël Guéhéneuc et al. [16] propose guidelines based on traditional design patterns. In short, constraints are specified which model a good implementation of a design pattern according to tradition. Detection of bad patterns is done by using these constraints to find out which areas of the given source code do not comply with traditional rules. These constraints are also associated with transformation rules which modify the given source code in order to comply with the constraint.

For this approach, a constraints solver is used, as each design pattern is specified using a number of constraints. A constraint is a set of \((\text{variable}, \text{domain})\) pairs. A variable may be anything in the meta-model such as a class, interface or association. The domain is the type of the variable (e.g. class, interface or association). A design pattern can be described using constraints by adding a list of conditions which must hold for the the design pattern to exist. For example, if a certain method must be public in a design pattern, then there would exist a constraint that says \text{public} is in the visibility modifiers of that method. Constraints are weighted by importance, and distorted design patterns are found by relaxing some constraints (so for example, we could relax the constraint that specifies the visibility of an attribute to see if a solution is found). This makes it slightly easier to find distorted design patterns without specifying all possible distortions (although it seems that all possible distortions would be considered automatically by the constraint solver anyway). A meta-model is defined to attempt to capture the semantics of the design pattern. The details of this have not been discussed, but a UML style representation is given. It consists of a collection of participant entities, each containing a collection of elements with different relationships between them. The meta-model is used to define an abstract model of the design pattern, which can be used to manually create the constraints listed above, which eventually are used to detect sets of entities matching the description of the design pattern.

Finally, Yann-Gaël Guéhéneuc et al. [16] present a tool prototype called PtideJ which generates the meta-model from source code, offers a graphical representation of the mapping, calls the constraint solver and uses the solutions to generate modified code using JavaXL. See Section 2.4.8 for more information on JavaXL.

The main assumption underlying Yann-Gaël Guéhéneuc’s work is that design patterns represent good architectures, and areas similar to design patterns represent potential points for improvement. This may be valid in many scenarios, however it makes it harder for domain specific knowledge to be encoded. Many applications have their own custom design patterns, tailored to their needs, and this is discussed further in 2.1.4. The approach proposed here seems to have certain design patterns embedded into the system, without the possibility of adding new ones easily. Writing constraints for design patterns is quite hard, especially for custom design patterns where there is likely to be a lot of variety in the types of constraints needed. The methodology proposed here also does not take into account any method to measure the degree to which the code has been improved, as it simply improves the code by making it conform to a pre-specified description of a traditional design pattern. Scalability should also be considered when using a constraints solving approach, as it seems that using such a meta-model and constraints set on a large application would be a very slow process. The meta-model that is specified here is not presented in depth, but seems to be similar to a UML diagram of the code.

In the context of this project, an idea similar to that of using constraints may be very useful. Design
2.3. Detecting Refactoring Requirements

Table 2.1: Types of data flow analyses

<table>
<thead>
<tr>
<th></th>
<th>Forward</th>
<th>Backward</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any path</td>
<td>Reaching Definitions</td>
<td>Live Variables</td>
</tr>
<tr>
<td>All paths</td>
<td>Available Expressions</td>
<td>Very Busy Expressions</td>
</tr>
</tbody>
</table>

patterns will need to be specified for the system to automatically analyse given source code in accordance with these patterns. This includes specifying traditional design patterns, as well as making it easy to specify newer design patterns. The latter was not considered in Yann-Gaël Guéhéneuc’s work, and is vital to this project. This design pattern specification will consist of multiple components for describing various parts of the design pattern. Variations in design patterns could then be spotted by analysing how much of the specification matches the given source code. More importance could be given to certain parts of the specification than other parts, in a similar way to the weighting of constraints performed by Yann-Gaël Guéhéneuc.

2.3.4 Static Analysis

Analysis of code can either be performed by executing the code, known as dynamic analysis, or by looking at the code without executing it, known as static analysis. This is a highly sophisticated approach, and has the ability to detect many of the code smells highlighted by Fowler \[14\], as well as some other indicators of inefficient coding. There are several advantages of performing static analysis instead of dynamic analysis. Firstly, it can be performed much earlier in the development cycle, and quite often more frequently in the development cycle than dynamic analysis can. This is because during development, different parts of the system are often not ready to be executed for testing purposes. Secondly, static analysis can be performed on a subsection of code, instead of the whole application. This again has benefits during development as the entire system is often not ready to be run. Thirdly, certain defects may be harder to find by executing the code. Such defects may even be defects in the logic of the code which don’t produce incorrect behaviour during testing, but will contribute to making the code harder to maintain in the future. Examples include redundant code such as duplicated or dead code.

Static analysis exists in two forms. The first consists of simple syntactic and other checks, most of which are already performed by modern compilers. Examples include checking for missing brackets and semicolons, and finding methods which don’t always return a value, or have potentially unreachable code after their return statements, and so on. The second and more important form of static analysis consists of finite state verification, well-formedness checks in specifications, and automated program analyses based on data flow methods. Some tools only consider the behaviour of individual statements, while others consider the behaviour of the program as a group of statements, in order to analyse whether the statements can be reorganised, or modified, in order to increase program efficiency or maintainability. Formal methods use static analysis to mathematically prove properties of programs in order to show they match the specification. These are usually restricted to systems where safety or security is paramount, such as in defense applications and air traffic control.

Some of the simpler static analyses are described here to give the reader a flavour of how static analyses work, and their power.

Data flow analysis

Data flow analysis is a technique for gathering information about the possible set of values calculated at various points in a program, using the program’s control flow graph \[44\]. There are four classic analyses quoted extensively in the literature. Table 2.1 summarises these.

- **Available variable analysis** - This analysis determines which expressions must have already been computed, and not modified later on, on all paths to that program point \[47\]. A definition is available if it can affect computation at the current location. An uninitialized variable will indicate an error. Available expression analysis is a forward flow problem, as the computed expression values
flow forward to points of possible reuse. The best solution is true, as the expression can be reused, and the worst solution is false, as the expression cannot be reused. Consider Listing 2.5. It should be clear that the expression \(a + b\) is available every time the execution reaches the while test, and so the expression need not be recomputed.

- **Reaching Definitions** - This is analogous to the previous one, except that we are now interested in determining which expressions may have already been computed and not modified later on, on some paths to that program point [47]. Similarly, it is a forward flow problem.

- **Very Busy Expressions** - An expression is said to be very busy at a program point if, no matter what path is taken from that program point, the expression must always be reused before any of its variables are redefined. Therefore, the aim of this analysis is to determine optimisations based on this information, and possibly store the very busy expression’s value for later use [47]. Consider Listing 2.6. The expressions \(a \times b + 50\) and \(a - b\) are both very busy at the start of the program, as they’re used on all points from it. Therefore, they may be hoisted out resulting in a reduction in code size, and a reduction in duplication of code.

- **Live Variable Analysis** - A variable is said to be live after a program point if there exists a path from that point to the use of the variable, and the variable is not redefined along the way. This analysis is used to determine for each program point, which variables may be live afterwards, and can be used as the basis for dead code elimination [47]. If a variable is not live after a program point, then if that program point defined that variable, it can be removed from the program. Listing 2.4 is an example where there is dead code which can be removed. This can be found using this analysis.

Each of these analyses aims to find some information for each program point. They can be solved using an iterative algorithm. It can be shown that this algorithm does terminate in a finite time, and its complexity is of quadratic order in the worst case, i.e. \(N^4\) where \(N\) is the number of lines in the program. This means that while the algorithm is solvable, it will take a reasonably long time for larger programs. This means that in the context of this project, it may not be possible to use analyses such as these to provide real time suggestions while the programmer is coding. It should however be possible to provide suggestions at regular intervals when the analysis is run. Another caveat to these analyses is that they are designed to work within a single method only. Inter-procedural variants of these analyses are possible and need more work on the part of the developer designing them. When considering inter-procedural
variants, there are a number of other factors to consider such as taking context into account, mapping variables onto parameters and so on.

However, dataflow analyses cannot be dismissed completely. They are a useful way of detecting problems in small blocks of code at the very least, and this project would benefit from a dataflow component which would aid the automatic refactoring system to not only detect and correct design flaws, but also detect and correct flaws in the low level coding of certain logic.

**Duplicate code detection**

Duplicate code can cause problems when the functionality needs to be changed, as the change needs to be replicated in a number of places. Thus duplicate code creates implicit links between methods, and even classes. Consider Listing 2.3. There is duplicate code here, and this needs to be extracted out into a function. Duplicate code is a common problem in many large scale programs, as developers wish to replicate existing code, and possibly modify it slightly. It can occur because developers are in a hurry and copy and paste code, or when two projects with similar code are merged into a single project, or simply when people forget that some code exists and rewrite it (the latter is often the case in large projects with several developers working on them over a period of several years). Duplicate code can be detected in a number of ways.

- **Line by line comparison** - Perform a line by line comparison and report duplicate code if lines differ only by whitespace. This is the least favourable method for obvious reasons. It can only detect code that has literally been copied and pasted. Often, “duplicate” code is similar to, but not exactly the same. It will fail to account for changes in variable names, or slight variations such as comments. Finding structural similarities is also very difficult when using this method.

- **Structural comparison** - Compare the AST (see Section 2.4.5) for structural similarities, in a similar fashion to the approach proposed by Baxter et al. [3]. Differences due to renamed variables, formatting and comments can then be disregarded to cater for non-exact matches. This method is more likely to work on larger systems with several classes, as it can cater for structure, meaning there are no inter-procedural problems. This means we may be able to detect two classes with similar functionality, or similar inheritance hierarchies. Structural similarities should be easier to deal with mechanically as the comparison is working at a higher level than the previous line by line comparison method. The trick with this method is finding a cut off point for defining when two fragments (or structures) of code are too different to be considered similar. This can be hard to do, as it requires some insight into the functioning of the program which is hard to provide automatically.

- **Algorithmically** - The Burrows Wheeler Transform and the Karp-Rabin algorithms are two examples of algorithms used to match duplicate strings in portions of text. Originally used for data compression purposes, these can be modified to detect duplicate code and provide a degree of flexibility in ignoring things like variable names and comments. The details of these are out of the scope for this report, but on a very high level, Karp-Rabin works by comparing the hash values of the code, while the Burrows Wheeler Transform tries to rearrange the code so as to group similar segments together.

It should be noted that all of these methods of duplicate code detection focus on very low level syntactic methods of detecting duplicate code. None are able to detect whether two fragments of code have the same behaviour, even if they wildly differ in syntax. For example, consider the problem of finding the factorial of a number iteratively and finding it recursively. The fragments of code doing this will be very different, yet they will produce the same result. It is nearly impossible to detect such behaviour duplication as that requires understanding of the aims of the code.

Another method for detecting duplicate code is described in considerable depth by Balazinska et al. [2]. This method determines detailed information about cloned methods and their dependencies, by performing a comparison at the lexical level (i.e. tokens) with the difference between two code fragments defined as the number of tokens that much be inserted or removed to convert one fragment to the other.
The comparison algorithm used is based on the Dynamic Pattern Matching Algorithm by Kontogiannis et al. [19], which finds the best alignment between two code fragments and uses statistical matching techniques. Matching is performed on vectors which are sequences of tokens forming the code fragments, and a grid is used to hold the detailed results of the match. The grid is a way of storing the numerical match distances between any two fragments of code, and the algorithm iterates through every possible combination of two fragments of code. This method has two immediately noticeable downfalls - firstly, it does not seem easy to find fragments of code which have been duplicated in more than two places, and secondly, it is very computationally expensive to compare every two possible combinations of fragments of code and determine the tokens that need to be added or removed from each one. The method has however been proven to work, and several instances of duplicate code have been detected in the JDK using this method. The cost grid is also useful when performing the refactoring of removing the duplicate code, as it tells you how much effort would be required for that refactoring.

For this project, duplicate code detection may be a useful addon. Duplicate code is one of the contributors to increasing complexity during software evolution. It often denotes a possible design flaw, and indicates the need for an Extract Method refactoring to be performed. An automatic refactoring system can utilize this to figure out possible candidates for the Extract Method refactoring to be performed. However, its utility is limited to this. Detecting duplicate code will not help the automatic refactoring system in any other way.

**Dead code elimination**

There are two kinds of dead code - assignments that are irrelevant, and code which can never be reached. See Listing 2.7 for an example of this. Modern compilers such as the Java compiler can detect the latter, but not the former. Live variable analysis has already been introduced as a means of eliminating dead code such as the former.

Like duplicate code elimination, the main caveat of this method is that it is very much focused on low level analysis, and does not consider a higher level abstract view of the program. In certain cases, the behaviour of the program may be identical with or without a certain statement, and therefore this statement would be dead code and could be removed. However, knowing the effect of all kinds of statements is not possible - some may be procedural invocations to libraries or code that is not available for analysis, while others may only produce visible behaviour when the environment the program is running in has certain characteristics.

This feature is another useful extension to this project, as it will help in the detection and correction of low level coding flaws within the program. It will not however help with any of the detection or correction of higher level design flaws.
2.3.5 Search Based Algorithms

A novel approach to detecting where refactoring needs to be carried out involves using search based algorithms. The most crude of these is the brute force algorithm which will try and apply every possible refactoring to every possible region of code, and see which one can improve the code by using some form of quality metric. This produces unsatisfactory results as firstly it is hard to define a quality metric to measure how good the design is, and secondly, results cannot be generated in a short period of time as every possible combination is being tried out - the time needed for larger systems will be much larger than the time needed for smaller systems, as the number of permutations to try will increase substantially.

The most commonly used form of search based algorithm which does not have some of the drawbacks of a brute force algorithm is a genetic or evolutionary algorithm. They were first presented by John Holland [18]. Genetic algorithms consider a population, where each individual is a solution to a problem. The solution is encoded in the “genes” of the individual. Genetic algorithms consist of 3 main steps:

1. Apply the rules of nature to select the fittest individuals - i.e. the ones that solve the problem in the best manner.

2. Cross-over individuals with each other. This is like reproduction in nature. The goal is that crossing over 2 good individuals should result in a new individual that combines the common good traits of its parents.

3. Mutation is also possible, so that children may have certain random characteristics, which quite often improve their genes.

The key components of any search based problem such as this is the representation of the problem, and the definition of the fitness measure which evaluates which individuals are fitter than others.

One of the characteristics of genetic algorithms is that they estimate solutions, but cannot prove conclusively that the generated solution is exact, or the best available. Genetic algorithms are different from heuristic methods in the sense that they work on a population of possible solutions, while heuristic methods work on a single solution and its iterations. Genetic algorithms are probabilistic, not deterministic, while most heuristic methods are deterministic.

The main advantages of genetic algorithms are that they can quickly scan a vast solution set, and that bad solutions don’t affect the final result negatively. This can be useful for loosely defined problems as well. The main disadvantage of genetic algorithms is that they are extremely slow, and become slower with a larger population. Another main downside is that they are liable to find a suboptimal solution, and there is no way to verify the optimality of a solution other than checking the fitness measure, which can often be quite subjective.

Harman and Jones [17] show how metaheuristic search techniques can be applied to the software maintenance and evolution domain. Metaheuristic search techniques are a set of genetic algorithms aimed at searching large spaces, which arise when there are a number of constraints which need to be balanced. Doval et al. [12] showed how such techniques can be used to decompose software systems into subsystems. Seng et al. [35] built on this work to propose an evolutionary search based algorithm for optimizing the class structure of a system. The algorithm proposed depends on metrics, and the number of violations of object oriented principles. Using metrics for refactoring has been discussed in further detail in Section 2.3.6. Seng’s approach is not fully automatic, but requires some initial user intervention to specify initial parameters.

Seng proposes a genotype, which is an “ordered list of executed model refactorings”, and a phenotype, which is created by “applying the model refactorings in the order given by the genotype”. There is a mutation operator which extends the current genome by adding more refactorings, and a crossover operator which combines two genomes. It does this by selecting some refactorings from the first parent, and adding those to the refactorings of another parent. The resulting set of refactorings is applied to the source code, and invalid refactorings are dropped. This is to guarantee that behaviour is not changed, unfortunately it looks like it would be a very time consuming process as the refactorings need to be simulated again.
The quality of the code after applying the refactorings is specified by a “fitness” measure, which is a weighted sum of several metrics which measure coupling, cohesion, complexity and stability. The code is allowed to evolve over a period of “generations”, and each evolution is assessed for quality. Seng notes that for a large project, about 2000 generations are enough to guarantee the optimal fitness value. Some small optimizations can be carried out in Seng’s work relating to attribute encapsulation methods - it doesn’t make much sense to move these methods, but the evolutionary process will include movement of these as part of the process anyway, unless otherwise marked. This marking process is manual, although could be automated fairly easily as encapsulation methods should fit a certain pattern most of the time.

The major problem with Seng’s work is that it is not intended to be run frequently on large pieces of software. A system with 30,000 lines of code took half an hour to analyse. This is likely to be the case for any search based algorithm. This is the major reason why this work cannot be of much benefit in the context of this project, which aims to develop an automatic refactoring solution which developers can invoke as often as they like during development. Another potential issue is the definition of the fitness measure. It is possible that the fitness measure does not accurately measure the quality of code in certain circumstances, and this is investigated further in Section 2.3.6. In addition to this, Seng’s genetic algorithm has the same disadvantages as any genetic algorithm, and these have been discussed previously. Another problem which is overlooked is that several design patterns deliberately violate design guidelines which are measured by the fitness measure. For example, the methods in a facade are low in cohesion as they forward incoming requests to other classes.

2.3.6 Metrics
Quantitative approaches using metrics can assist developers in detecting potential design flaws and finding useful restructurings to correct them [34]. Indeed, if such restructurings could be automated with a quantitative and formal basis, design quality and development productivity could be improved [21] as well as monitored and compared with other designs.

Chidamber and Kemerer [5] proposed the first theoretically sound set of metrics of measuring the quality of software. A brief overview of these metrics is presented below, along with their advantages and limitations, to give the reader an idea of how to measure software quality with metrics.

- **Weighted Methods Per Class (WMC)** - Sum the individual complexity values of each method. The ratio of this sum and the number of methods indicates the average complexity and therefore possibility of reuse and maintainability. The authors have not defined how to calculate the complexity value of a method, and claim this is a strength as it makes the metric generic. It could also be viewed as a weakness, as there is no formal basis on which to compute the complexity. This metric may be used to indicate where methods have become too complicated, and could indicate the necessity for an “Extract Method” refactoring (see Listings 2.1 and 2.2).

- **Depth of Inheritance Tree (DIT)** - The depth of inheritance of a class, or the maximum length from superclass to subclass given multiple inheritance. The theory is that the deeper the class is in the hierarchy, the more methods it will inherit, making it more complex to manage. Once again, this may only point out areas where design may be improved, but does not present any concrete way to improve the design.

- **Number of Children (NOC)** - The number of immediate subclasses of a class. The basis for this comes from the fact that a larger number of subclasses indicates greater possibility of reuse, i.e. a better design. Unfortunately, this may be self contradictory as a larger number of subclasses may also indicate a greater likelihood of inappropriate abstraction. Therefore this metric is not very suitable for judging quality.

- **Coupling Between Object Classes (CBO)** - The count of the number of other classes to which a class is coupled. This is calculated by looking at all classes that are referenced by a class. Excessive coupling can be hazardous to design and prevents reusability. Therefore this figure should be minimized in most cases (although some design patterns encourage coupling for various reasons).
• **Response For a Class (RFC)** - The number of methods that could potentially be executed by methods of a class. This may be used as a complexity measure for the class, as the larger this number is, the greater the complexity of the class. Once again, this metric only potentially indicates problems, not conclusively.

• **Lack of Cohesion in Methods (LCOM)** - This is a count of the number of method pairs whose similarity is zero minus the count of the number of method pairs whose similarity is not zero. Similarity is defined by the number of instance variables that are used in both methods. A high LCOM value will indicate that a class is trying to achieve many different objectives (low cohesion), while a low LCOM value will indicate that a class is trying to achieve a highly defined set of objectives, or possibly even just one objective (high cohesion). This notion of similarity is quite simplistic, and it could be argued that two methods might use the same instance variables but achieve completely different objectives and therefore even a low LCOM value is not guaranteed to mean that the design is good.

Lee and Wu [21] extended Chidamber’s [5] approach and stated that a long term research goal is to define transformations that preserve behaviour, and are automatable with metrics. In other words, Lee and Wu attempted the use of metrics to perform refactorings. They claimed that cohesion and coupling metrics can be used to quantify designs and compare one design over another. In their approach, restructurings are defined using eight primitive non-behaviour transforming operations, and built up using these primitives in order to validate preservation of behaviour. It can be shown that the primitive operations defined are sound, complete and minimal (i.e. every operation is design restructuring, the set of operations subsumes every possible restructuring, and is the minimal such set to do so).

It is common design practice to maximize cohesion and minimize coupling. This reduces the likelihood of change to objects, and therefore the consequences of change, as well as increasing the reusability of objects. The cohesion and coupling metrics defined by Lee and Wu in [21] are based on abstract models known as call-use graphs and class-association graphs, along with communication matrices between methods. The call-use graph shows methods, attributes and their connections representing call relationships and use relationships. The class-association graph shows classes and the connections between them, representing inheritance and interaction relationships. The communication matrices store the amount of information communicated between methods, excluding constructors, destructors and accessors. The defined metrics combine characteristics of these graphs and matrices defined to generate a number which is a good representation of how good the design is. This approach is good in the sense that it does not rely on the detection of certain patterns in the code, rather it relies on certain characteristics of code which are likely to be present in any good design. Relying on spotting known good patterns in code may be misleading, as people often define new patterns suited to the particular context.

A fitness function is used by Lee and Wu [21], just like [35]. The fitness function here uses the cohesion and coupling metrics defined, along with information about inheritance depth. A genetic algorithm is applied using the fitness function. There are disadvantages to this, namely that genetic algorithms take a long time to find an optimal solution, and are often not too good at fine tuning around local optima. This has been discussed further in Section 2.3.5. However, the fitness function, metrics and operations defined here are valuable, as minimizing the fitness function while increasing the cohesion metric and decreasing the coupling metric will likely (although not necessarily) indicate a better design. These metrics could be used to quantitatively analyze the quality of a design generated by any method, assuming the design doesn’t deliberately violate object oriented principles to achieve another goal (such as efficiency).

To summarize, metrics have the potential to be very useful for the development of an automated refactoring solution. They can indicate places where the design may be flawed, and in certain cases, they may indicate which refactoring needs to be applied. However, it should be noted that the key word in the previous sentence is *may*. Metrics are not guaranteed to indicate problems in design, nor are they guaranteed to indicate which refactorings will improve the design. Often, coupling and cohesion are necessary given the context, but metrics may flag them up as poor design. Even if the metric was correct in identifying excessive coupling and low cohesion, it cannot indicate how to go about performing a refactoring, and so the metric will not help much in the context of this project which aims to automatically implement refactorings to improve the design.
2.4 Currently Available Tools and Frameworks

In this section, several tools and frameworks are discussed which help in the refactoring process. Some of these tools provide developers with a way of automatically refactoring code, provided the developer makes a decision about which part of the code to refactor, and which refactoring to apply. Others highlight potential design deficiencies, and make suggestions about implementation methods. The rest provide base functionality which is essential to any system that will be modifying source code in a programmatic fashion (i.e. automatically).

2.4.1 ArgoUML

ArgoUML is an open source UML modelling tool, and is compliant with the OMG standard for the UML 1.4 specification. It has an implementation of the Java Metadata Interface, and uses the GEF Framework to provide a graphical view of the UML diagram. It can reverse engineer code into class diagrams, given a Java project. The class diagrams can be edited, and the Java code regenerated given the new diagrams.

In terms of design analysis and refactoring, the key feature of ArgoUML is the built in Design Critic. It is a set of threads which continually monitor the design, and suggest improvements. These improvements include simple things such as syntax errors and style guidelines, but support for more complicated improvements is under construction. The design critics help to identify areas of the model that do not obey simple design rules or best practices. The results are presented in a To-Do list, and a wizard is available to help implement the suggestions made. Currently the critics available in ArgoUML are:

- **UML critics** - These look for things like classes that have too many methods, which would make them harder to maintain (i.e. trying to avoid god classes).

- **Cognitive critics** - These are supposed to look for critics related to design pattern flaws. Currently, they only support the singleton pattern.

- **Java critics** - These are related to Java specific issues. Currently, they only support warning against modelling multiple inheritance.

These critics produce suggestions based on a series of heuristics, which can be thought of as a set of rules that are applied by a rule engine to the state of a model. The ArgoUML API is fairly extensible, and creating a new critic involves adding a class to the `org.argouml.uml.cognitive.critics` package which extends `org.argouml.cognitive.critics.Critic`. Each critic should define its own predicate method, which performs the analysis of the design. Critics implementing this abstract method are expected to follow certain rules like type checking objects passed to them (UML entities which model the design). Each object in the design is expected to register its critics with the run time system. Methods are available to add items to the To-Do list and generate wizards to help implement the suggestions. The implementation of a critic must also involve some form of parsing of the given UML entity to check for whatever it is that the critic wishes to check for. How this is done is left up to the developer of a new critic, and there does not seem to be any concrete way to do this. This makes the process of writing a new critic quite cumbersome, as a whole array of methods must be implemented. Existing code in the `org.argouml.uml.cognitive.critics.Init` class must also be changed to cater for the newly developed critic. This adds to the tediousness of making new critics, and also suggests that the design of ArgoUML critics architecture has not been properly planned out. The documentation for making new critics is also very poor.

While ArgoUML is heading along promising lines, it seems that at the moment, not much support is available for automatically detecting many design flaws apart from the very basic ones. ArgoUML does not support any way of implementing the design flaws that have been highlighted, but does offer wizards to guide the programmer through the implementation process. ArgoUML is still quite heavily concerned with the implementation level analysis of the design, and has critics to monitor class selection, naming, design patterns, inheritance, circular composition, illegal generalization etc. It does not provide any higher level guidance for analysing the design itself to cater for intended design goals. The critics that ArgoUML has are not interrelated - in other words, they do not interact with one another to find
the best solution. Instead, the critics all work independently to produce suggestions. This is not the best method, as a human expert would combine all of his or her knowledge before making suggestions. From this, it can be seen that any tool developed as part of this project should have the ability to consolidate all of its existing knowledge to form a solution.

2.4.2 PMD

Code reviews are conducted regularly in large projects, and widely recognized as being very effective ways to detect inefficient code, security issues, incorrect business logic, incorrect use of programming principles etc. The main aim of code reviews is to improve code quality, and PMD is a static code analysis tool that focuses on preemptive defect detection [36] in order to help in the code review process. It is capable of automatically detecting a range of defects such as non-optimized or unsafe code, dead code, overcomplicated expressions, duplicate code and possible bugs such as empty try/catch/finally/switch statements. It should be noted that PMD only works on Java code, and no attempt has been made to make it language independent.

PMD can integrate into existing development environments such as Eclipse, allowing for real time analysis of code. It categorises suggestions into five levels of severity, and displays them in a list for the programmer to review. This categorisation is not automated, but must be entered by the developers of the rules for PMD. This means that the developers of the rules of PMD must have a shared understanding of what each severity level means, otherwise the whole concept of prioritization using severity levels becomes irrelevant. PMD does not provide a way to implement corrections to the problems that have been detected.

Programmers can choose to ignore certain suggestions made by PMD either by dismissing them from the tasks list, or by adding annotations to the code. For example, @SuppressWarnings("NameOfRule") will suppress the rule called NameOfRule from being used to analyse the following piece of code.

PMD works by using the JavaCC parser generator (see Section 2.4.5) along with JJTree to generate an AST by parsing the code. This AST can be traversed using the Visitor design pattern.

Creating new rules for use in PMD is easier than making new critics in ArgoUML. PMD provides two ways of writing rules:

1. **Write a rule using Java**
   A new Java class should be made which extends net.sourceforge.pmd.AbstractRule. PMD generates an AST for the code, and traverses it recursively providing callbacks to rules that are interested in the types it encounters during the traversal. Therefore the newly created Java class should implement a visit method which will get called whenever the AST traversal finds a node of a certain type. Once the rule has been written in Java, it should be added to a ruleset XML file - this file contains all of the rules available to PMD, along with descriptions, and it maps them to the Java code.

2. **Write an XPath expression**
   A more concise way to write rules for PMD is to use XPath expressions. The XPath expression will be applied to the AST, and matching nodes will be returned for analysis. For example, if we wish to say that no variables should ever be instantiated as Object (i.e. Object a = new Object()), then we can find all such instantiations by the rule //AllocationExpression[Name/@Image='Object']. The primary advantage of XPath expressions is their conciseness. Adding new rules using XPath expressions does not involve any recompilation of Java code - only XML ruleset files must be updated.

PMD is better at finding design defects than ArgoUML because it has a wider range programmed into it, and is also much easier to extend to add new rules. Unfortunately, adding new rules to PMD involves studying the structure of the AST in order to see how code fragments are defined in tree format. Like ArgoUML, PMD provides quite a low level view of the coding defects, and does not cater for any higher

---

6PMD - http://pmd.sf.net. The developers themselves don't know what PMD stands for
7JJTree - an add-on to JavaCC which decorates the parser with an AST
null pointer dereferences

```java
Car a = new Car();
a = null;
System.out.println(a.toString());
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Listing 2.8: Null pointer dereferences</td>
<td></td>
</tr>
</tbody>
</table>

null pointer dereferences

```java
this == null => false;
this != null => true;
$a && false => false;
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Listing 2.9: Jackpot rules</td>
<td></td>
</tr>
</tbody>
</table>

level design flaws, perhaps to do with the way object oriented principles have been used. The closest it gets is by using coupling rules, which find instances of inappropriate coupling by counting the number of unique attributes or imports in a class, and setting a threshold level to indicate high coupling.

### 2.4.3 FindBugs

FindBugs\(^8\) is a similar tool to PMD in that it finds bugs in Java programs. However it differs in its internal workings - while PMD works on the code level, FindBugs works on the bytecode level. It statically analyses the Java .class files by matching bytecode against a list of known bug patterns. In addition to this, and unlike PMD, FindBugs has a dataflow component to perform dataflow analysis. This lets it find a wider variety of bugs such as null pointer dereferences. Consider the example in Listing 2.8. This will obviously generate a runtime exception, even though it will compile fine. FindBugs can detect this error by performing dataflow analysis. PMD will not be able to detect such errors. This shows that dataflow analysis (see Section 2.3.4) is a vital component of any code defect analysis tool such as the one being developed in this project. Unfortunately, as FindBugs works on the bytecode level, it can be harder to define custom rules, as that requires familiarity with the Java bytecode specification.

### 2.4.4 Jackpot

Jackpot\(^9\) is a Netbeans IDE plugin which enables the reengineering of Java source code. Examples include converting code to no longer use deprecated methods, or simplifying overly complex conditional code. In a similar fashion to PMD, these changes are made either with custom Java transformation classes, or via rules files which use a custom Java pattern matching language. Jackpot works by using information made available by the Java compiler itself, and using the rules files to pattern match against the AST generated by the Java compiler. This is a unique approach, as the other transformation tools all use custom AST representations of the code. It is claimed that this approach enables Jackpot to have more detailed and correct information about the code.

A Jackpot rule has two parts to it - the pattern to be matched (before the ⇒), and its replacement (after the ⇒). It is possible to use metavariables to refer to parts of the pattern to be matched, if these parts need to be reused in its replacement. Listing 2.9 shows two examples of rules which simplify redundant code (note that this can never be null in Java). Note that $a is a metavariable in the last rule, and could be used in the replacement section if it was required (e.g. we could say $a && false => $a if we wanted to).

One important aspect of Jackpot is its ability to chain transformations together. Unlike ArgoUML where the critics all run independently of one another, several Jackpot rules can be applied automatically onto the same piece of code. Consider the example in Listing 2.10. Here, several rules are applied one after the other to obtain the simplest possible representation of the code.

---

\(^8\)FindBugs - [http://findbugs.sf.net](http://findbugs.sf.net)

\(^9\)Jackpot - [http://jackpot.netbeans.org](http://jackpot.netbeans.org)
2.4. Currently Available Tools and Frameworks

Listing 2.10: Jackpot example transformation

```java
    do {
        someMethod();
    } while (k && this == null);
```

becomes

```java
    do {
        someMethod();
    } while (k && false);
```

becomes

```java
    { // do $a; while(false) => $a;
        someMethod();
    }
```

Unlike ArgoUML or FindBugs, Jackpot is really easy to extend. As the example above shows, its pattern matching language is fairly simple, and is capable of doing a reasonably large number of transformations to the source code. Such a language will prove very useful in the context of this project, but additional features would be needed which can transform more of the design elements of the code. At the moment, Jackpot is highly focused on syntactic transformations. While Jackpot is easy to extend, and it is possible to add new transformations pretty easily, Jackpot only works with the Netbeans IDE. This reduces its usefulness slightly. Another important aspect of Jackpot is its ability to automatically correct the defects that have been detected. Other tools generally output the defects in a list, for the programmer to consider. Jackpot has the option of correcting the defects automatically.

The ideas behind Jackpot will be very useful for this project. Jackpot has an easy to understand syntax which makes it easy to extend, and the same should be the case for the automated refactoring solution being developed in this project. Jackpot’s concept of specifying a “bad” coding pattern and its corresponding “good” coding pattern is very useful for specifying certain kinds of flaws in code which cannot possibly be observed by an automated system otherwise. Thus, a similar approach to this project would be handy.

2.4.5 Parsers and Parser Generators

Central to any code analysis system is the analysis of code, and to do this requires the source code to be parsed, and some form of structure generated which can represent the source code. The most basic form is the Abstract Syntax Tree. See Section 2.6.1 for ways of generating a more semantically oriented form. An abstract syntax tree (AST) is a “finite, labeled, directed tree, where the internal nodes are labeled by operators, and the leaf nodes represent the operands of the node operators” \[43\]. It is often used as a compiler’s internal representation of a computer program during code optimization and generation. In an abstract syntax tree representation of the code, variables and constants will appear as the leaves of the tree, while constructs like assignments and method declarations will appear as nodes. A number of parsers are discussed below which can parse code and generate these ASTs from them. Note that the internal workings of a parser is beyond the scope of this research.
JavaCC

JavaCC \(^{10}\) is a parser generator and works by using grammar files, that specify the language features, to create a recursive-descent Java parser which can recognize source code conforming to the grammar specification.

The parser generated by JavaCC can be used to parse given source code, and in doing so, an AST is generated. To get an idea of what this AST looks like, an experiment was conducted. Listing 2.12 shows the Java source that was used, and Listing 2.13 shows the AST that was generated.

JavaCC is intended to produce parsers for any language, be it real or custom or toy. Unfortunately, the grammar files for real languages are not published by the language developers (e.g. Sun). They are supplied by the open source community, and are not guaranteed to conform to the Sun specifications. This means that the grammar files cannot be relied upon to generate a fully accurate parser and AST.

Antlr

Antlr \(^{11}\) is a tool which provides a framework for constructing parsers automatically for a language, given a description of the language in EBNF \(^{12}\). Antlr uses the EBNF descriptions to generate recursive-descent parsers. It can create lexers and parsers in any language, although Java seems to be the language of choice.

The parser generated by Antlr can be used to parse given source code, and in doing so, an AST is generated. To get an idea of what this AST looks like, an experiment was conducted. Listing 2.12 shows the Java source that was used, and Listing 2.11 shows the AST that was generated. Note that this was using JavaEmitter - this is a class that abstracts the Antlr AST to produce a slightly more readable version.

Just like JavaCC, Antlr is intended to produce parsers for any language - this includes toy languages, custom languages, and real languages. Unfortunately the EBNF specifications for real languages are not published by Sun, the developers of Java. The EBNF specifications for the Java language in Antlr are written by the open source community, and this will inevitably mean that there are mistakes in them, or that they do not conform to the latest language specifications. This means that using JavaCC or Antlr for this project will be quite difficult.

Eclipse

In its simplest form, Eclipse \(^{13}\) is a generic IDE for any language, but primarily meant for use as a Java IDE. It is open source, and its core is made up of the JDT \(^{14}\). The JDT core is what adds the features of a Java IDE to Eclipse. Most importantly in the context of this project, the JDT core contains a Java Model that provides API for navigating the Java element tree (AST) generated from Java source code. The Java element tree defines a Java centric view of a project - it builds a semantic view of the code by creating elements like package fragments, compilation units, binary classes, types, methods and fields, which can all be accessed programatically. For example, the nodes in the AST which refer to method and variable declarations are referred to as MethodDeclaration and VariableDeclarationFragment. The JDT also contains an annotation processor, intended to support the processing of annotations in Java source code. The JDT API contains comprehensive support for source code modifications by performing a rewrite of the generated Java element tree, or AST. It also supports modification of code at a higher level, e.g. adding classes, interfaces and methods etc. Figure 2.2 \(^{20}\) summarises the typical workflow of using an AST to modify Java source code in Eclipse.

It has to be noted that the JDT only works on Java code. The alternative for C++ code is the CDT \(^{15}\). This provides C++ support within Eclipse. Similar to the JDT, it too builds an AST representation of the C++ code, which can be modified using a fairly comprehensive API.

\(^{10}\)JavaCC - [https://javacc.dev.java.net](https://javacc.dev.java.net)
\(^{11}\)Antlr - ANother Tool for Language Recognition - [http://www.antlr.org](http://www.antlr.org)
\(^{12}\)EBNF - Extended Backus Naur Form
\(^{13}\)Eclipse - [http://www.eclipse.org](http://www.eclipse.org)
\(^{15}\)CDT - C++ Development Toolkit - [http://www.eclipse.org/cdt/](http://www.eclipse.org/cdt/)
2.4. Currently Available Tools and Frameworks

Listing 2.11: AST generated by Antlr using the JavaEmitter

```plaintext
text:AST ROOT type=0
text:package type=16
text:ANNOTATIONS type=50
text:javatest type=68
text:CLASS_DEF type=14
text:MODIFIERS type=5
text:Test type=68
text:EXTENDS_CLAUSE type=18
text:IMPLEMENTS_CLAUSE type=19
text:OBJBLOCK type=6
text:METHOD_DEF type=9
text:PARAMETERS type=20
text:PARAMETER_DEF type=21
text:OBJECT type=13
text:METHOD type=9
```

Listing 2.12: Sample source code used to generate an AST

```java
package javatest;

class Test {
    public static void main(String args[])
    {
        System.out.println("Hello_world");
    }
}
```
Listing 2.13: AST generated by JavaCC

```
CompilationUnit
  PackageDeclaration
    Name: javatest
  TypeDeclaration
    ClassOrInterfaceDeclaration (Test) (class)
      ClassOrInterfaceBody
        MethodDeclaration (public) (static)
          ResultType
          MethodDeclarator: main
          FormalParameters
            FormalParameter (package private)
              Type
              ReferenceType
                ClassOrInterfaceType: String
              VariableDeclaratorId: args
          Block
            BlockStatement
              Statement
                StatementExpression
                  PrimaryExpression
                    PrimaryPrefix
                      Name: System.out.println
                    PrimarySuffix
                      Arguments
                        ArgumentList
                          Expression
                            PrimaryExpression
                              PrimaryPrefix
                                Literal: "Hello world"
```

Figure 2.2: Abstract Syntax Tree Workflow in Eclipse
Eclipse itself uses the JDT to perform its semi-automatic refactoring. This refers to refactorings such as Pull Up and Push Down method and Move method as described in Section 2.2. These refactorings move methods and attributes up and down an inheritance hierarchy. The programmer can right click on a relevant field or method, and choose a refactoring to perform from the menu. It is for this reason that the refactorings should be referred to as semi-automatic - the programmer has to select both the region and refactoring of interest.

When performing a Pull Up refactoring in Eclipse, a warning is given to the user about the fact that attributes in the subclass cannot be accessed from the superclass, even by using a cast. Ignoring this warning and performing the refactoring results in a program which compiles, but has a known antipattern - superclasses referring to subclasses. This is an unwanted consequence of Eclipse’s refactoring. Another inconsistency appears when performing the Move Method refactoring in Eclipse. This happens when methods with the same name, but different code are moved from different classes to their common superclass. If the methods to be moved to the common superclass differ from each other only as a result of casts being used, then these casts will appear in expressions that access attributes declared in the subclasses. Eclipse does not generate a warning about the fact that the method bodies are different, and this can lead to behaviour inconsistency, or a break in the functionality. Pushing down a method using Eclipse’s refactoring tools can also lead to errors in certain scenarios, as Eclipse does not give any warning when we push down a method to subclasses, excluding a subclass whose instances are target of calls to the method we are pushing down [8]. This results in a compilation error.

Despite these inconsistencies within the Eclipse refactoring framework, the JDT is an excellent way of generating ASTs from Java source code. The JDT is guaranteed to conform to Sun Java specifications, and provides very good support for editing of code, and regeneration of code based on the edited AST, as shown in Figure 2.2. This makes it a very good framework to use for this project, which is going to deal with the editing of given source code according to certain specified criteria. Having a framework to perform the required editing would be very useful, as more development time can be spent analysing what editing to perform rather than how to perform it. In order to best utilize a large framework like the JDT, a wrapper might be required which makes available just the functionality that is required for this project.

2.4.6 Transmogrify

Transmogrify is a Java code analysis tool which is capable of performing refactorings on the code. It has not been developed for nearly 6 years now. It works as a plugin for Borland’s JBuilder and Sun’s Forte4Java. These IDEs will generate a parse tree for the source code. After this, the programmer must choose the region that he wishes to perform a refactoring on, and select the appropriate refactoring from the Transmogrify list. The available refactorings are very basic, and similar to the ones available in Eclipse today, such as Extract Method and Pull Up Field. Thus, Transmogrify does not offer any new ideas in terms of functionality, and it is a fairly standard semi-automated refactoring tool for Java.

Transmogrify works by translating the source code into an AST, using Antlr (see Section 2.4.5). In order to create a new refactoring, the developer must get the the cursor position from the IDE and determine which AST node has been selected. This AST node can then be modified programmatically in memory, before being rewritten out as source code. This is a very messy process for two reasons. Firstly, getting the cursor position to determine which region of code to refactor makes this framework highly dependent on the IDE, and reduces reusability severely. Secondly, the programmer who is using these refactorings will find it very tedious to have to look through an AST to find the region of code he wishes to refactor. Eclipse handles this much better by allowing the programmer to right click on a portion of code in the text editor itself. Therefore, any tool developed in this project should offer easy integration with popular IDEs in order to avoid tedious of use.

16Transmogrify - http://transmogrify.sf.net
Listing 2.14: Sample source code used to test JavaXL

```java
public class Test
    /* Random comment */
    implements AnotherClass {
}
```

Listing 2.15: Source code used to make changes using JavaXL

```java
// This finds the class called Test in the source
XClass test = new javaXL.XClass("Test");

// This adds a superclass to the Test class
test.setSuperClass("YetAnotherClass");
```

2.4.7 OpenJava

OpenJava \(^{17}\) can be thought of as a toolkit for constructing a Java preprocessor. It provides an API which programmers can use to handle source code and perform translations on it at compile-time. It uses the Java Reflection API to do this. The result of this approach is that programmers cannot see the output of the transformations that have been performed on the code. This is not desirable behaviour when creating a system which automatically corrects design deficiencies in code. However, this is not the main aim of OpenJava. Its goal is to provide programmers with a way of extending the Java syntax to let them add compile time checks to the code, almost like a preprocessor. While this does not solve the problem of automatically detecting defects and correcting them, it helps programmers to identify defects which they themselves are checking for. This required conscious effort on the part of the programmer is what reduces the usability of this framework.

2.4.8 JavaXL

The Java eXtended Language, or JavaXL, is intended to modify user source code to comply with design pattern descriptions [1]. The main motivation behind JavaXL is to apply a design pattern onto an existing implementation by modifying existing code as little as possible. This helps to preserve developer specifics such as coding conventions and comments. Consider the example in Listing 2.14. JavaXL could be used to modify that as shown by Listing 2.15. This will result in Listing 2.16.

Unlike OpenJava, JavaXL does not perform a complete source re-generation, and this ensures that it can preserve as much as possible of the existing code. Unfortunately, JavaXL only ensures that the source code is syntactically correct, and does not check for semantic correctness (i.e. it does not check for any behavioural or design defects). Another difference with other source to source transformation

\(^{17}\)OpenJava - [http://openjava.sf.net](http://openjava.sf.net)

Listing 2.16: Changed source code generated by JavaXL

```java
public class Test extends YetAnotherClass
    /* Random comment */
    implements AnotherClass {
}
```
2.5. Refactoring Techniques

Listing 2.17: Program Slicing

```java
performSomething1ForTask1();
performSomething1ForTask2();
performSomething2ForTask1();
performSomething2ForTask2();
performSomething3ForTask1();
performSomething3ForTask2();
```

Program slicing is a technique first suggested by Weiser [40]. It is a method used to abstract semantics from programs. It can be used to identify those portions of a program which influence the behaviour of a collection of designated statements. Consider Listing 2.17. Lines 2, 4 and 6 may be considered to be a single slice which can be used to perform an “Extract Method” refactoring. The same applies to lines 1, 3 and 5. In other words, slicing works on the fact that only a portion of the program’s behaviour is of interest. This is the case when a subset of the behaviour is being either corrected, maintained, updated or replaced. The “slice” produced is an independent program, guaranteed to represent the original behaviour. This can be particularly helpful when trying to extract portions of behaviour from a system, in order to simplify the system by making it more modular. This may be required as too much excess functionality has been built into one area, as requirements grew over time.

Once a program slice has been taken, it can be optimized and re-inserted into the program. Program slices may also be defined as regions of the system with design flaws. This behaviour of interest is called a “slicing criterion”. Weiser suggests that data flow analysis (see Section 2.3.4) should be used to find this behaviour based on the defined criterion. This works by finding all statements that actually affect the value of a variable at a certain point. There are three main problems with this approach. Firstly, the notion of a slicing criterion is not clear. It seems that one has to specify a very large number of criterion to cater for all possible cases of interest (i.e. those cases where a design flaw may be evident). Secondly, the problem of finding a slice is unsolvable [40]. It is not feasible to expect an automated program to be able to understand the semantics of the source code and figure out how to slice the program to find regions of interest. Data flow analysis may help, although it is not a guaranteed solution to finding related statements and ensuring that the statements in between them can actually be moved. Lastly, program slicing is meant to work more on the code level rather than the design level. This is partly because at the time that the technique was suggested, object oriented programming was not as popular as it is today. While this technique may be useful for performing optimizations at the code level, there are much better formal techniques available for such static analysis and optimization of code (as described in Section 2.3.4). It is for these reasons that program slicing will not be considered a viable solution in the context of this project.
2.5.2 Graph Transformations

Using the process of graph transformation, the software is visualized as a graph, and restructurings occur in the form of transformation rules. Software entities like classes, variables and methods are represented on a graph as nodes. The edges between the nodes represent the relationships between them, such as inheritance, method calls and variable accesses [25]. This approach abstracts away from language specific features, and represents the core object oriented design. In order to fine tune the graph representation, Mens suggests the use of “well formedness constraints”. These are enforced using “type graphs” and “forbidden subgraphs”. The latter, as the name suggests, looks for subsections of the graph which are known not be valid representations of syntactically correct programs. The former is a meta-graph, which expresses restrictions on the contents of the instance graphs. Enforcing correctness of graphs using constraints is a tedious procedure, as it involves specifying a list of constraints for a program to be correct. It is similar to the constraint solving techniques discussed in Section 2.3.3.

A “graph rewriting” [25] is a transformation that takes an initial graph as input, and transforms it into a result graph. This occurs using pre-defined transformation rules. Mens did not specify many such transformation rules, and it seems to be the case that this approach is not very generic. Design flaws often don’t occur in their purest form - i.e. variants of flaws normally occur. The transformations given here are very specific, and often may not match the correct starting conditions, as they cannot cater for variations design flaws. Another problem is that specifying constraints for anything but the most simple of transformations is highly complex, especially for nested structures, which frequently occur in large systems. Mens has also not yet considered combining graph transformations together to form a sequence of refactorings, and this is looked at in further detail by Roberts [33]. Thus, graph slicing seems to offer quite a verbose and complicated way of performing transformations, and it seems hard to define new or custom transformations using this approach. Therefore, this technique will have limited utility in the context of a project aiming to develop a customizable automated refactoring solution.

One important feature of graph transformations is that they have are formal enough to prove that program behaviour is preserved. Mens showed that the graph rewriting formalisation preserves certain types of relationships such as updates, accesses and invocations [26].

2.5.3 Strategy Based Elimination

Trifu [39] points out that “there is no clear understanding of exactly what a design flaw is”. Today’s state of the art in design flaw elimination is missing a vital link between problem detection methodologies and correction techniques [39]. To account for this, Trifu tried to introduce a concept of correction strategies, which are human-made descriptions which outline ways to remove design flaws. Trifu outlined a set of criteria which any design flaw elimination system has to fulfill.

- C1: High level of abstraction - language independent approach
- C2: Quality driven - take into account the impact of the transformations on the quality of the system
- C3: Causality - don’t apply all possible refactorings in the hope of finding one which improves the system the best, but select the best possible refactoring to apply
- C4: Scalability - A good methodology should be just as effective in solving low level and high level flaws
- C5: Behaviour presentation - Transformations should preserve the behaviour of the system
- C6: Extensible - The approach should be able to cater for other design flaws too
- C7: Automation - The methodology should enable substantial automation
- C8: Dissemination - The tools should be able to record and transmit expertise.
2.5. Refactoring Techniques

Trifu’s correction strategies attempt to satisfy these criteria, and are based on the concept of nodes and branches. Nodes are major refactoring decisions which impact the system quality, whereas branches are alternative paths which result as a consequence of choosing a node. The choice of which node to take depends on human intervention, and hint levels which the developer must assign. Although after a while the tool can use past decisions to make new decisions on its own, this approach significantly reduces the automation aspect of the tool, as quite often a developer will want to eliminate design flaws without having to do a lot of work. Trifu describes an example correction strategy for fixing large conditional structures in code, using an extended BNF notation. There is however, no concrete implementation of such a language.

Trifu admits that it’s not yet clear how the algorithms for estimating the best refactoring solution would look like. The quality model in use by the proposed methodology is essentially a “tree that allows the decomposition of targeted quality factors into software metrics”. Thus, the quality of the quality model would depend entirely on the software metrics, and it is widely believed that defining software metrics accurately and generically for any possible systems is not possible. Trifu noted that his correction strategies don’t represent a program, or indeed any algorithm which can be used to eliminate design flaws. They are just a description of a solution and its variations to a given flaw, along with some quality related information. This does need an additional methodology to specify how to make decisions using the correction strategy information, and this has not been elaborated upon.

Despite these flaws, Trifu’s approach is a good starting point, and the criteria defined by Trifu are indeed useful for the evaluation of any possible approach that may be taken by this project.

2.5.4 Composition and Dependencies between Refactorings

When creating complex refactorings, it is important to determine which refactorings are mutually independent and which need to be applied sequentially. The independent ones may be applied in parallel, which speeds up the refactoring [11]. This is a useful concept to keep in mind in the context of this project which seeks to help developers during the development as well as maintenance processes. Mel Ó’Cinnéide [28] presents a method for developing composite refactorings for Java programs in such a way that it is also possible to demonstrate behaviour preservation. An algorithm is described for design pattern transformation and composes refactorings using sequencing, iteration and conditional constructs. Primitve refactorings are defined which have pre and postconditions. For example, the addMethod refactoring would add a method to a class iff a method with that signature doesn’t already exist. M. Ó’Cinnéide describes the two main methods of composing refactorings and presents a technique for deriving the pre and postconditions of a composite refactoring.

- **Chaining** - This is where a sequence of refactorings are applied one after the other. For example, the chain in Listing 2.18 adds methods foo and foobar to the class c.

- **Set Iteration** - This is where a refactoring, or refactoring chain is performed on a set of program elements. For example, the set iteration in Listing 2.19 copies all the methods of one class to another.

Note that a chain is the key component of any refactoring composition, and it may be of any length. However, the computation can be simplified by observing that the problem only needs to be solved for chains of length 2, as the procedure can be repeatedly applied to the remaining chain until the full pre and postconditions have been computed. The precondition of the chain can be calculated by looking at the preconditions of the individual chain components, and ANDing those preconditions together which are not guaranteed by the postconditions of the previous chain element. During this, contradictions may
arise, and so the chain would be invalid. Similarly the postcondition of a chain can be computed by
ANDing together the postconditions of the individual chain components.

O’Cinnéide describes a useful framework for composing refactorings together with the chaining and
set iteration methods. No methods to implement this framework have been specified, and it seems
like manual intervention is needed to define the pre and postconditions. Examples of typical pre and
postconditions are sketchy, as are the details of the theory. No internal representation of the program
has been considered, and analysis and helper functions are used to derive information about the program.
The main contribution of this work is the method to compute the overall pre and postconditions of a
refactoring statically, for later reuse. When creating an algorithm to analyse when certain refactorings are
necessary, such a technique may be useful for this project. However, this technique will only identify if a
particular refactoring can be applied, and not whether it is suitable or necessary to apply this refactoring.

2.5.5 Behaviour Preservation

When a refactoring is performed, it is ideal if it can be shown that the refactoring will not have any
unwanted consequences. Thus, the refactoring needs to be behaviour preserving. There are two ways
of looking at this - the refactoring needs to either preserve the behaviour of the code, or it needs to
preserve the intent. The latter is obviously much harder as it involves analysis of the semantics of the
code, and this can only be achieved by taking annotations from the programmer. This renders this
approach infeasible in the context of this project which seeks to automate the refactorings by keeping
user interaction to a strict minimum. Thus, only those approaches which can preserve the behaviour of
the code are presented here.

Another aspect that needs to be considered is what kind of behaviour preservation is required. Some
parts of the behaviour are less important than others [11]. For example, real time systems depends
on the execution of certain sequences of operations. Safety critical systems require that properties like
liveness and safety are preserved. Embedded systems have power consumption and memory constraints
as essential aspects of behaviour. Often, these kinds of behaviour can only be analyzed at runtime using
benchmarking tools. This shows that the field of behaviour preservation is very wide, and depends hugely
on the domain involved.

Mens et al. [27] consider notions of behaviour preservation which can be detected statically, and do
not need any data or control flow analysis techniques to be employed. Thus, three forms of behaviour
preservation are identified, and ideally a refactoring should conform to all of these to be considered as a
behavior preserving refactoring.

- **Access Preserving** - A refactoring is access preserving if each method accesses at least the same
  variables after the refactoring as it did before (transitive access is also allowed).

- **Update Preserving** - A refactoring is update preserving if each method updates the same variables
  as it did before the refactoring.

- **Call Preserving** - A refactoring is call preserving if it performs at least the same method calls as it
  did before the refactoring.

Mens shows how certain refactorings, such as Pull Up Method, is behaviour preserving according to these
categories. These are a very simplistic set of forms of behaviour preservation, and indeed other items
could be added to this such as type preservation. This form of static analysis may give a good indication
that a refactoring was behaviour preserving. However, this method will not guarantee a correct analysis

### Listing 2.19: Set iteration as a method of Refactoring Composition

```java
forall m: Method, className(m) = a {
    addMethod(m, c);
}
```
each time. Some refactorings may deliberately remove accesses or updates to variables or methods, and according to this method, that would not be a behaviour preserving transformation. This brings around the initial argument of whether the transformation really should preserve behaviour or whether it should preserve intent - some behaviour is unwanted, and refactorings are removing that behaviour.

Lee and Wu [21] defined a set of eight primitive operations on their abstract code models which were shown to be sound, complete and minimal. This was discussed further in Section 2.3.6. O’Cinnéide [28] mentions support for multiple languages by presenting a prototype tool which shields language specifics using a layered architecture. She states that this is analogous to the fact that designers of a system have a much higher level, language independent view of the system, and refactoring tools should share this view. The methodology proposed is not a fully fledged way of automating the detection and correction of design flaws, rather it is a way of introducing design patterns into a system, once the correct patterns have been identified. The methodology is similar to composite refactorings (Section 2.5.4) in that it:

- Decides on a precursor for the transformation - this helps determine where the transformation begins and ends.
- Decomposes the transformation into a sequence of mini-patterns. These are design motifs that occur frequently [28].
- Develops a minitransformation corresponding to each mini-pattern. This is a set of pre- and post-conditions, and a sequence of transformation steps. These are the most important unit in this tool, as they can be re-used over many different mini-patterns, and design patterns.
- Defines the design pattern transformation as a sequence of minitransformation compositions.

The methodology proposed is sound, and also guarantees behaviour preservation. However, the developed tool has only been tested on Java so far, and that again on a limited number of design patterns. The scalability of the architecture proposed is yet to be investigated, and it is possible that that not all of the transformations that are required could be decomposed into minitransformations. While the concept itself is good, more work is needed to investigate the extensibility of this architecture to cater for new varieties of transformations.

For the purpose of this project, a simple notion of behaviour preservation such as in [27] will be suitable in the earlier stages. For several reasons as discussed above, it will be impossible to guarantee behaviour preservation, and quite often we want the behaviour to be changed anyway. Lee and Wu’s [21] primitive operations for transforming programs will be useful for the implementing of transformations in this project, as the operations defined are very basic ones and have been shown to be useful in creating larger transformations.

2.6 Language Independent Refactoring

One area of current research focuses on defining refactorings in a language independent manner, so as to enable refactoring tools to serve any existing languages and integrated design environments. One of the key advantages of this is to be able to share domain knowledge between languages. Such knowledge could include being able to detect and correct common design errors, or introduce patterns to make the design better. O’Cinnéide’s [28] contributions to this field have already been discussed in Section 2.5.5.

2.6.1 Meta-modelling

Having a model of the code seems to be an essential part of language independent refactoring. Tichelaar et al. [38] conduct a feasibility study for a language independent refactoring engine. They propose a meta-model for describing refactorings at a highly language independent level, and a prototype tool to refactor both Smalltalk and Java.

The primary goal of the meta-modelling approach proposed by Tichelaar was to support the most common aspects of languages, rather than all aspects of all languages. This approach views systems at
the “program entity level” rather than the abstract syntax tree level. This is a crucial step in minimising
ganguage dependence. The control flow within methods is not relevant here either, allowing for a suffi-
ciently abstract analysis. While Tichelaar claims that this approach is sufficient to represent the most
primitive refactorings on a language independent level, he also acknowledges that higher-level refactorings
are outside the scope of the study. This project will focus on higher level refactorings as well as lower
level ones, and so Tichelaar’s approach may be of little help.

A FAMIX model [10] is proposed here, which provides a representation of object oriented code. The
core model specifies classes, methods, attributes, invocations and accesses, and the relationships between
them, in a similar way that a UML diagram would. FAMIX was part of the FAMOOS Information
exchange project, aimed at developing a “reengineering method for transforming object oriented legacy
code into frameworks”. The FAMOOS reengineering life cycle is fairly simplistic, and involves require-
ments capture and analysis for model definition, problem detection, resolution, reorganisation and change
propagation. This is a promising framework, but work on it ended in 1999.

Tichelaar’s refactorings are fairly basic, and involve adding and renaming methods and classes. Pre
and postconditions have been listed for each refactoring, but these only indicate if it is feasible to perform
a certain refactoring, rather than which refactoring to perform to improve design - this choice is left to
the user of the tool.

Tichelaar et al. [38] showed that language independence was possible to an extent with the use of
FAMIX, but that not all language differences could be abstracted from. A few such examples are included
here:

- Whether or not certain variable/method/class names are valid for a particular language.
- Static and dynamic typing [23] of languages is a problem. Some languages (e.g Java) are statically
typed while others (e.g. Smalltalk) are dynamically typed. Dynamic typing reduces the amount of
information available at compile time, and makes reengineering a lot harder. However, C++ and
Java are both statically typed, and they are probably by far the most commonly used languages.
- Object oriented features vary between languages - some languages may support multiple inheritance
while others do not. Modelling this in an abstract manner is not possible.

It should be noted that bringing language independence into a refactoring framework has certain
overheads additional to the ones mentioned by Tichelaar:

- The meta model becomes overly complicated when dealing with a model which caters for all lan-
guages - certain features may be found in the model which are irrelevant for the codebase being
studied.
- The meta model is language independent, and refactoring may possibly be able to transform this
into a better model. However, this model needs to be converted back into source code to be of any
practical use. The changes to code will always be language specific, and therefore the changes on
the model will need to be mapped onto the code. Language specific information may also need to
be stored for the different entities involved.
- Some refactorings may be beneficial to one language but not the other. This brings another layer
of complexity.

Piefel [30] attempts to address some of these issues noted by Tichelaar by generating an intermediate
model tailored for code generation. This model contains behavioural as well as structural aspects of the
code, and is applicable to Java and C++. These languages are chosen because they are very common and
similar to each other. However, there are differences in visibility and multiple inheritance, and therefore
the metamodel suggested can only represent the intersection of features from both languages.

Piefel’s metamodel is called CeeJay, and uses a package with common metamodelling building blocks
as shown in Figure 2.3 taken from [30]. At the root of the metamodel is an abstract ModelElement and
each element in the model is a subclass of it. NamedElement adds the ability for a model element to have
a name, while Namespace is an abstraction for either package in Java or namespace in C++. Note that
in Java and C++ the concept of packages or namespaces is slightly different - Java requires packages to be placed in appropriately named directories, while C++ namespaces can be arbitrarily included into compilation units. However, these differences are not important in this metamodel and this is one of its advantages. Note also that in the model presented in Figure 2.3 has no constraints which limit any model elements to be contained in packages or namespaces. This is analogous to both Java and C++ where classes can exist out of packages.

CeeJay does have a few disadvantages, namely that the specifics of Java and C++ seem hard to model. Examples include things like missing keywords (e.g. `virtual` is missing in Java), and lack of operator overloading support in Java. Therefore, CeeJay is good as a high level representation of the program structure, but needs more work to model the specific details of the languages. Lack of preprocessing ability in Java is another aspect which may be hard to model - unless the C++ preprocessing was completely ignored in any metamodel which is not an ideal solution.

Language independence will be a useful feature to have in this project, which is based around developing an automated refactoring tool to help in the maintenance and development processes of any software system. It would increase the scope to which the developed tool will be of use in existing major projects. Piefel’s approach is a useful starting point, and shows how a model may be developed. For this project, it would be useful if a certain popular language, such as Java was the core focus, but language independence was kept as a secondary priority. Thus for any models being developed in this project, the implications to language independence should also be considered. CeeJay’s approach of finding similarities between languages (such as packages in Java and namespaces in C++) may be replicated in this project, in order to form a basic level of language independence.

### 2.7 Summary of Background

After studying the current state of the art in this field, it is possible to make some observations about where the main problems lie. Current development environments and tools offer a very narrow view of the source code, often at the code level rather than the design level. The transformations available are very basic ones, and often limited to predefined transformations. It is often not possible to combine these
transformations together. Several tools were developed with poor documentation, or have been outdated as a result of lack of development. From the developers point of view, the process of refactoring consists of:

1. Identifying problems
2. Identifying appropriate refactorings
3. Applying the identified refactorings

Modern day tools (e.g. Eclipse) only offer support for the last of these steps, leaving the first two to the programmer. Some tools (e.g. ArgoUML) offer support for the first step and leave the last two steps to the programmer (while offering limited guidance). No tool is available which integrates all of these steps together to form a fully automated refactoring solution.

As highlighted in Section 2.4.3, any such system must contain a dataflow component. Extensibility also seems useful, as its lack in tools (e.g. ArgoUML) proved to be their major downfall. Useful tools such as (e.g. PMD) were found to be easy to extend (Section 2.4). Jackpot (Section 2.4.4) was found to be especially easy to extend, and offered both detection and correction facilities. However Jackpot only caters for very basic transformations. It is therefore desirable that this system is easy to extend and can include harder problem detection and correction facilities, possibly with custom design patterns.

It was found that tools like PMD are quite code focused, and therefore the some code-level correction of flaws would be beneficial. However, it was also found that many tools are overly focussed on the code, and not on the design and so this system needs to incorporate some design level analysis too. It was found (Section 2.4.6) that certain tools like Transmogrify use very tedious ways of performing refactorings, and still don’t cater for the entire refactoring process. Therefore this system should aim to make the refactoring process as seamless as possible by integrating into existing development environments thus reducing the necessity for extra software to be installed. Utilising existing knowledge about development practices also seems like a useful addition to this project, as it is not a commonly found feature in existing work in this area. It was found that tools like ArgoUML suggest improvements to the code without consolidating all of their knowledge, resulting in possibly contradictory suggestions, or suggestions which don’t make the best possible use of the available information. It is for this reason that this project should aim to produce a solution which can consolidate all information given (in the form of design patterns, antipatterns and so on) in order to produce a refactoring which takes into account a variety of factors.
Chapter 3

Overall Design

“Computer language design is just like a stroll in the park. Jurassic Park, that is”
- Larry Wall

Key to any project is the understanding of what is required for successful completion, in other words - the goals. This chapter will commence by looking at some of the goals for this project, and some potential challenges that each may create. This will lead into a discussion about the design decisions made, and finally an explanation of the overall structure of CoderAider, its various components, how they were created and how they interact with each other.

3.1 Aims

Firstly, we must consider the factors which have influenced the architecture of CoderAider - the aims. These are broadly derived from studying the capabilities and limitations of existing tools, as discussed in the previous chapter, and will drive the evaluation of the completed system. It is important that there are two categories of users of CoderAider. Firstly, there are end-users - these are developers using this project to analyse and improve their code structurally. Secondly, there are refactoring designers - these are developers using this project to specify new refactorings which can be applied by CoderAider onto existing projects. These two categories of users may overlap, and end-users might decide to write their own code transformation schemes to suit domain specific purposes. We need to take this into consideration when designing CoderAider.

- Interaction with existing development practices - A refactoring tool will likely be widely adopted if it is compatible with existing development practices. Ideally, compatibility with an IDE is required, as programmers can then apply the functionality immediately without further configuration or installation of packages. The key point to realise here is that there exist a large variety of different development practices and environments, and it will be difficult to ensure compatibility with all of these. Therefore, we need to at least ensure that compatibility can be achieved for a reasonable subset of existing practices.

- Minimal human intervention - As we saw in the Introduction, human intervention is often a factor that dissuades developers from using so called “automated” refactoring tools. We should attempt to keep human intervention to an absolute minimal for this tool. Developers should not need to organise or annotate their source code in special ways in order for this tool to function. This poses the challenge of having to automatically find the areas of the code which need to be refactored. Most existing tools do not offer this functionality, and rely on the user to specify which refactoring to apply, and which area to apply it on. Overcoming this hurdle will be a key achievement for CoderAider, as it will open the way for a whole new line of research relating to integrating the process of detecting defects, and fixing them using the appropriate correction.
• **Detection & Correction** - The system should be able to do all of the following things - identify problems in code, identify an appropriate solution for each problem, and apply that solution to the code.

  - The system should be able to use specifications of refactorings to detect areas of code which have a high probability of being flawed, or inefficient. The key difficulty to overcome here is that of false positives - many existing tools provide results which a human would deem to be irrelevant, and reducing these should be a goal.

  - These specifications should be easy to create, especially from existing codebases of older projects, as knowledge gained from them is likely to be the major source for any refactoring specification. This will make CoderAider better than existing refactoring toolkits, as they are not designed with this in mind. It will however be challenging, as existing projects may contain various different kinds of “knowledge” which we wish to apply to newer projects.

  - Once detected, the system should be able to either suggest an improvement and implement it, or highlight to the user how best to proceed with correcting the flaw. We must take care here to select the appropriate improvement to apply, if there is a choice of more than one. Existing tools ignore this difficulty by forcing the user to choose which improvement to apply - it is this interaction which reduces the utility of existing code improvement tools.

  - When implementing these improvements to code, it is desirable for the system to be able to guarantee that the behaviour of the code is preserved, or that the behaviour is modified in such a way that is guaranteed to make it better. The main aspect of achieving this aim is defining how behaviour can be programmatically determined - after all, the halting problem states that it is impossible for an algorithm to determine if another algorithm will terminate - so CoderAider will have a hard time determining whether behaviour, even for simple things such as termination, is guaranteed to not be modified by any code transformations performed. We therefore need to look at alternative means of behaviour preservation, and perhaps look to involve the user in some way to help guarantee this.

  - It is essential that the system preserves any coding conventions which the programmers have been using, along with other information such as comments. This is important to ensure that programmers do not lose familiarity with their own code, as a result of programmatic reformatting, or deletion of comments.

  - The system should attempt to ensure syntactic correctness, and attempt to ensure that compilation errors do not arise as a result of changes made. The key thing to note here is to ensure that if a sequence of transformations are being made, then conflicting transformations do not alter the code in a way that might make it syntactically invalid.

• **Operational aspects**

  - It should be possible for the user to select which refactorings this tool should apply. This selection is only an added feature, but not an essential part of the refactoring process.

  - The tool should also highlight in some form the changes that it has made to the user’s source code, or present a preview of the changes. This will help towards building confidence in the developer that the tool has not modified parts of the codebase without logging or highlighting the changes made.

  - The tool should enable the user to selectively apply the suggestions made.

  - Once the changes have been applied, the user should still be able to Undo the changes as well, all in one go. Needless to say, performing an Undo should revert the code back to its original state.

• **Flexibility** - There should be a reasonable degree of flexibility when it comes to the types of refactorings which can be catered for. If the system is designed properly, it should also be possible to use the system for means other than those originally intended. The system should be flexible
3.2. Interaction

CoderAider aims to provide users with suggestions for areas of code that may be improved, as well as suggestions on how to improve potentially problematic areas. The primary focus is Java, and Eclipse is a common IDE for Java. Therefore, it is designed as a plugin for Eclipse. Details are discussed in Chapter 4. The user must select which types of suggestions to analyse for, and the user must select which package or set of classes to analyse. This serves as the input for CoderAider. Figure 3.1 shows the overall architecture of CoderAider. This will be explained in following sections. For now, note that CoderAider takes details of the source code from the workbench, performs analysis and transformation on them, and rewrites updates source code to the workbench.
Figure 3.1: Overall architecture
3.2. Interaction

Figure 3.2: Lifecycle of a refactoring

3.2.1 The lifecycle of a refactoring

Firstly, let us define how the process of performing refactorings in CoderAider should work, as shown in Figure 3.2:

Step 1 The refactoring is started by the user by selecting an appropriate class or package, and right clicking. An initial quick check is performed to determine whether the refactoring is applicable in the context desired by the user (CoderAider makes itself available at the package and class contexts only).

Step 2 The user is asked for additional information if necessary (Figure 4.10 is the actual screenshot). Here the user can select which code examples (refactoring specifications) to apply. The user can now press Finish to perform the analysis (Section 3.3.2) and transformation (Section 3.3.4) and go to Step 4, or press Preview and go to Step 3.

Step 3 The proposed changes are shown to the user in a preview dialogue, at which point the user can either choose to apply the changes, ignore them altogether, or partially apply the changes (Figure 4.11 is the actual screenshot). The differences are shown in a diff style, so that the user can see the old and new code at a glance.

Step 4 The selected changes are applied to the workspace using the Rewriter component (see bottom of Figure 3.1).

Step 5 Optionally, the user can undo all of the changes made in a single Undo operation.

Step 6 Optionally, the user can make some changes to the source code, and repeat from Step 1.
3.2.2 Reasons for supporting Java

Java is one of the most popular languages these days, and many large software projects have been created in Java. Aside from this, Java offers some key benefits to us when creating a refactoring system:

- It is strongly typed. This makes it easier to analyse the code statically to determine the hierarchy of classes and interactions that occur between them. It also makes it simpler to define refactoring specifications such as the ones discussed in the next section, as well as modify source code.

- There is significant support available in existing frameworks and IDEs for tools which seek to improve Java code. This includes frameworks such as the JDT which provide models of source code which can make it easier to analyse and modify.

- Java is a typical implementation of an object oriented language. We should take care to separate the object oriented features from Java specific features when creating a refactoring framework, so that future work may be carried out to extend the scope of this project to include other languages such as C++ or C#.

3.3 Performing the Refactoring

Between Steps 2 and 3 of the lifecycle of a refactoring (Section 3.2.1), CoderAider performs its major work. Figure 3.3 shows a brief overview of this. Each step will now be explained.

3.3.1 Step 0 - Refactoring Specifications

CoderAider aims to provide a method for performing refactorings that is highly extensible, and capable of using existing projects to draw upon for knowledge. As such, this can only be achieved if there is a catalogue of available refactorings to which users and/or refactoring designers can add new refactorings. Therefore, the Refactoring Specifications (Figure 3.1) are an important part of CoderAider. A Processor will process each refactoring specification at runtime, and pass the results to a Model Generation process, described shortly. The details of the refactoring specifications are discussed in Chapter 5. Essentially, they are a set of code examples which show some “bad” code, and its “good” equivalent. Alternatively, they can be a code example which shows some desired structural goal. Annotations are used to add flexibility to this specification system. The main hurdle that this solves is allowing developers to create
new specifications with ease. They can either write their own code examples, or use existing projects to find examples worth using - thus solving (partially at least) the goal of allowing knowledge from existing projects to be transferred to newer projects.

We must define some limitations to the types of examples that may be written here. Support for the synchronized, volatile and transient keywords is present, but is very basic. These keywords modify the behaviour of the code in ways which can be quite subtle. If the example “good” code includes these keywords, they will be added to the user’s code too in order to make it look like the example “good” code. However, no tool checking will be performed to ensure behaviour or semantics are preserved for these keywords, as a deadlock or a race condition may be introduced by use of these keywords, and the issue of multithreaded program behaviour falls outside the scope of this project. It is assumed that the refactoring designer that wrote the example code for the refactoring specification is aware of the implications imposed by the introduction of these keywords. This highlights an important assumption - the refactoring designer which creates the code examples for use in CoderAider is aware of the improvements that his examples will make and knows about any potential side effects they may have. In this way, we also mitigate the difficulties of behaviour preservation to an extent, as we rely upon the refactoring specification designer (who provides examples of “bad” and “good” code) to ensure that the examples are correct.

To summarise, Step 0 (of Figure 3.3) deals with defining and using some example “bad” and “good”. Some of these definitions will be included with CoderAider, and developers are encouraged to write their own ones for their own domain specific purposes, or from their existing projects.

Code Model

In order to analyse or transform the source code supplied by a user, CoderAider needs to define a code model on which to operate. The Eclipse infrastructure does a lot of the groundwork for this, by providing lexing and parsing capabilities, and by providing an Abstract Syntax Tree (AST) of the code. The model generated by CoderAider incorporates this AST model, but adds a layer of functionality around it intended to simplify operations involving code analysis and transformation. This defined model acts as a semantic wrapper for the AST as it defines the code from an object oriented perspective rather than as nodes with children. Chapter 6 delves deeper into the specification and generation methods for the code model, and explains the rationale behind this design.

The model defined here can be used for several purposes (thus demonstrating the flexibility of CoderAider), such as:

- **Code analysis** - in order to find regions for improvement, using specific code examples to drive the detection process.
- **Code rewriting** - through the application of a Transformer and Modification Logger (see Figure 3.1) which tries to ensure behaviour preserving (syntactically and semantically) valid transformations.
- **Code evaluation** - through the computation of metrics, which may be used to further help the code analysis and rewriting functions. Chapter 9 provides more details about this.
- **Code criticising** - in order to highlight areas of code which are thought to have a defect which the system is unable to fix automatically.

It should be noted that both the source code of the user, and the refactoring specifications (which is just example code) is processed internally by using the same Model Generation process to create models. This is shown in Figure 3.1. The refactoring specification models are augmented with the additional information found in the code annotations, which was added to provide a greater degree of freedom while specifying refactorings.

### 3.3.2 Step 1 - Matching

An important step of refactoring is finding the potential regions of the user’s source code where refactorings are applicable. This is done by use of a Matcher phase (see Figure 3.1 and Figure 3.3), which takes
the model of the refactoring specifications (code examples) and attempts to find similar patterns in the user's source code. Matches are stored in a Match Set, which can be passed along to the next stage in the transformation process. Further details about matching can be found in Chapter 7. In essence, the algorithm will work by going through each refactoring specification and attempting to match the “bad” example listed with some part of the user’s source code. Attributes, methods, constructors, superclasses and superinterfaces will be considered when performing this matching. This method can help in identifying which parts of the user’s code is seen as “bad” by a refactoring specification, i.e. which parts are prime candidates for refactorings to occur.

During this phase, it becomes clear that the aims outlined earlier are slightly contradictory. On one hand, we said that we want to avoid false positives (i.e. make matching as precise as possible), and on the other hand, we said that the refactoring specifications should encompass a whole class of refactorings (i.e. enable matches to find many results). While satisfying both of these goals, there is inevitably a trade-off, and finding the right balance between preciseness (fewer matches) and variance (encompassing a class of refactorings, i.e. more matches) is the key challenge. This is overcome using an algorithm which assigns weightings to different factors within the code examples, in order of importance. Chapter 7 explains how this works.

To summarise, Step 1 (of Figure 3.3) deals with using the supplied “bad” code examples to find areas of the user’s code that are similar to these.

### 3.3.3 Steps 2 & 3 - Transforming the source code

The next logical step is using the Match Set (areas where refactorings may potentially be applied) to carry out some refactoring. This is done by comparing the “bad” and “good” code examples in the refactoring specifications to determine how to transform the code in order to make it the same as the “good” example. One the sequence of steps have been determined, the same steps will be applied to the user’s source code too, to transform it into code which matches the example “good” code. This is the function of the Transformer (Figure 3.1). Details of this process are in Chapter 8.

As noted earlier, the key difficulty in performing transformations is ensuring that they do not compromise the state of the code by either making it syntactically invalid, or introducing faults into it. To accomplish this objective, each transformation is given to a Modification Logger (Figure 3.1). This has the job of logging the modifications to be made to the user’s source code, updating the internal model of the code and most importantly, monitoring the consistency of transformations and ensuring the integrity of the source code by editing unfavourable transformations (i.e. those that are unrequired, or may cause compilation issues later). This also makes it simpler for the Transformer to function, as it merely as to output a stream of transformations (as found from the code examples) without worrying about their validity or potential for conflict. More details of this are given in Section 8.2. Note that this logger will help in the implementation of refactoring change previews as well, whereby the user is shown a preview of the refactoring before it is actually applied (Step 3 of the lifecycle outlined in Section 3.2.1).

To summarise, Step 2 (of Figure 3.3) deals with finding the differences between the “bad” and “good” code examples provided, and determining what changes to make to the “bad” code so that it looks like the “good” code. Step 3 applies these changes onto the user’s code, so that it looks like the “good” code.

### 3.3.4 Step 4 - Rewriting the source code

Once the transformation phase has completed, the Modifications Logger possesses the list of changes which must be made to the user’s source code. This list is passed on by the Transformer to a Rewriter (final stage of Figure 3.1) which passes through the user’s code and makes the changes listed. This is a very mechanistic process, and much of this functionality is provided by the Eclipse Refactoring Toolkit. The updated source code can then be written back to the user’s editor in the Eclipse IDE, or presented for preview to the user before being accepted. Details are given in Sections 4.2.1 and 4.3.1.

There are several reasons why we have decided to leave this stage till the very end.

- We can perform multiple analysis and transformation passes through the source code before deciding upon a final set of transformations to make. This helps to leave the user’s code intact during this
process, which is especially useful if the Transformer later decides that it no longer wishes to perform a certain transformation.

- It becomes easier to encapsulate changes into one operation, meaning easy reversal if required. Changes are also grouped together instead of being spread out over many operations, meaning the user has a better idea of what has changed as a result of this particular invocation of the analysis. This also helps when logging the changes made.

To summarise, Step 4 (of Figure 3.3) saves the changes made to the user’s code to the user’s editor, so that the user may proceed with development.

3.4 Summary

In this chapter we have described the aims behind CoderAider, and presented an overview of its design, discussing some challenges faced along the way, and highlighting the rationale behind decisions taken in order to overcome these challenges. To recap, Figure 3.1 shows the overall architecture of CoderAider. CoderAider works as an Eclipse plugin - it takes source code from the workbench, and generates a custom model of it using the Model Generation process. The Model Generation process also works on the refactoring specifications (examples of desired coding practices, or examples of poor practices and how to improve them). The generated models are compared for matches, and the results form a Match Set. This is passed to a Transformer which figures out the transformations needed, and applies them to the user’s source code through a Modifications Logger, and a Rewriter. Figure 3.3 shows the details of these last few steps, namely that the “bad” code examples are matched against the user’s code to form a Match Set, differences are analysed between the “bad” and “good” code example to find out what changes to make the “bad” code, these changes are applied to the user’s code (in the Match Set), and the changes are written back to the source file.

The next chapter will present the implementation details of CoderAider itself, and how it functions inside Eclipse. Screenshots will be given, and some implementation difficulties will be highlighted.

Chapter 5 will outline the strategy behind letting developers specify code examples for desired or undesired practices, and why the chosen methodology should outperform existing tools of a similar nature. Chapter 6 will then focus on the code model that was designed, and explain the rationale behind its design. Chapter 7 will describe how we can match the provided code examples (refactoring specifications) to existing pieces of code, and Chapter 8 will look at how transformations can be made to the code model, in order to satisfy the refactoring specifications provided. Finally Chapter 9 will outline how we can use metrics in conjunction with the analysis and transformation phases to heuristically improve the results.
Chapter 4

Eclipse Infrastructure

“In a way, staring into a computer screen is like staring into an eclipse. It’s brilliant and you don’t realize the damage until it’s too late.”

- Bruce Sterling

An aim of this project was to develop an automated refactoring solution which was as simple to use as possible. The main problem that we are attempting to solve is to make any methodology developed by CoderAider as easy to integrate into existing development environments as possible. As the scope of this project is restricted to Java, we decided to opt for integration with Eclipse, an IDE widely used for Java development. Eclipse provides a rich array of features suited for our purposes, and how these were used will be briefly highlighted in this chapter. This chapter will also give a brief overview of the Eclipse plugin contributed by this project, and will help to put the rest of this report into context. The internal workings of this plugin are discussed in following chapters, and explain the novel ideas contributed by this project.

4.1 Plugin Framework

Eclipse provides us with a Plugin Development Environment (PDE) for rapid development of Eclipse plugins. A plugin is the smallest extensible unit in Eclipse. Plugins have extension points, which let other plugins interface with them. Formally, they are named entities for collecting contributions. A plugin can contribute new functionality to the workbench by adding an extension to extension points declared by other plugins.

Figure 4.1 shows the organisation of the various components of Eclipse. At the core of the Eclipse platform is the Workbench and Workspace. The JDT is part of the Eclipse framework, and is implemented as a plugin to the Eclipse Platform. The PDE is another plugin to Eclipse, and hooks into the JDT. We are using the PDE to develop this project. CoderAider is an Eclipse plugin which extends the functionality provided by the Eclipse refactoring features (which in turn are implemented as plugin). As a result of this, it is possible for other plugins (called “Other Tool” in Figure 4.1) to extend the functionality provided by CoderAider, but it is also possible for CoderAider to utilize the functionality already provided by the existing Eclipse refactoring framework. This makes the decision to implement CoderAider as an Eclipse plugin a worthwhile one.

Implementing an Eclipse plugin also has the advantage that developers that have Java projects setup in Eclipse can use the plugin without further configuration. In fact, Eclipse allows for very simple installation of new plugins by use of the automatic updater.
4.2 Eclipse Java Model

The Eclipse framework provides a Java Model that provides API for navigating the Java element tree. The Java element tree defines a Java centric view of a project. It surfaces elements like package fragments, compilation units, binary classes, types, methods and fields\(^1\). Building a model of given source code using this framework proved to be the best way of dealing with the problem of analysing programs. Alternate methods of model generation such as javacc and Anlr were evaluated, but none of these had any significant advantages over using the JDT core, and indeed most lacked the detailed integration with the Eclipse core which was needed for this project. Figure 4.2 [20] shows the way that Eclipse organises Java elements in a hierarchical manner, making it easier to generate a custom model for the project.

We have decided to concentrate refactoring efforts on a level higher than that provided by current tools, i.e. on the class level and package level. Functionality relating to one aspect of the project is usually found in one of these two levels, so there is no need for CoderAider to run on any level higher than this. Therefore, the decision was taken to make the functionality of CoderAider available at the IPackageFragment and ICompilationUnit level. As shown in Figure 4.2, these correspond to the Eclipse representations of packages and classes respectively.

---

\(^1\)JDT Core - [http://www.eclipse.org/jdt/core](http://www.eclipse.org/jdt/core)
4.2. Eclipse Java Model

As an example of the power of the Eclipse JDT, Figure 4.3 [20] shows the information that it makes available for a method declaration. Note the breakdown of method modifiers, name, return type, parameters, thrown exceptions, body and so on. This pool of information about the source code is what makes this an ideal choice for code analysis in this project. This information is used to build a custom model of the source code, and this is described further in Chapter 6.

4.2.1 Making modifications to the Eclipse Java Model

One of the aims of CoderAider was to ensure that any changes made to the user’s source code will not adversely affect his/her familiarity with the code (by either reformatting it, or removing comments etc). This was one of the main driving forces behind deciding how to make modifications to the Eclipse Java model.

In order to make modifications to the AST, two alternatives are available. The AST can either be modified directly by use of the AST editing API provided by the JDT, or an instance of ASTRewrite can be used to record a list of modifications to make to the compilation unit (class AST) without actually making the modifications, thereby leaving the AST intact. Essentially, it implements a tree rewriter on ASTs. It offers methods to declare insertions, removals, moves or copies of AST nodes. An AST node may be a field, a method, or a variable name, a parameter, a statement, a block of code, or even a visibility or other modifier. Figure 4.4 shows the lifecycle of using an ASTRewrite object to handle rewriting of code. The source code is taken, an AST is generated, nodes of this AST are either deleted or modified, and the changes are rewritten back onto the textual source code.

It is this ASTRewrite approach that is used by CoderAider. This is because:

- Refactorings can be logged.
- Multiple ASTRewrite objects can be created for each AST depending on refactoring requirements.
• Changes that will be made to the current AST are all visible easily. If the AST was modified directly, it would not be possible to quickly find out which changes have been made by the current CoderAider refactoring.

• It becomes easier to create an encapsulated object storing the changes to be made to the source code. This is important when using the LTK, described in the next section.

Note another approach to making modifications is to generate the entire AST from scratch, however this is a computationally expensive operation, and one that will not preserve code formatting conventions used by the developers. Therefore, this approach was not considered for this project.

4.3 Refactoring Framework

Eclipse has a Refactoring Language Toolkit (LTK) which supports automated Java refactorings. It includes an infrastructure which enables other plug-ins to take part in refactorings selectively via extensions.

This refactoring framework provides the necessary infrastructure to contribute a refactoring to the refactoring history, the refactoring scripting facility and the Eclipse workbench itself. Refactorings can be reliably executed on a local workspace, with the LTK taking care of the details related to precondition checking, change creation and change validation. The refactoring framework user-interface provides abstract implementations of refactoring wizards, refactoring input pages and will show precondition checking errors and change previews [41].

In summary, some of the features of the LTK that were used in the development of CoderAider are:

• Refactoring User Interface - The LTK provides a user interface which enables the creation of a refactoring wizard which provides facilities to enter the necessary input used to initialize the refactoring object.

• Precondition Checking - The LTK provides a framework for basic precondition checking such as asserting an error-free workspace and valid user input. CoderAider uses this to ensure that the analysis is only performed at either the class or package level in the project. CoderAider also uses the precondition that the code being analysed is syntactically correct, and compiles properly. Analysing errors which can already be caught by the Java compiler is not part of this project.

• Refactoring History Integration - The LTK helps in seamlessly integrating all refactorings with the Eclipse refactoring history. CoderAider ensures that executing its refactoring results in a refactoring descriptor which is persisted in the global refactoring history by the refactoring framework.
4.4 CoderAider integration into Eclipse

We will now briefly demonstrate how CoderAider has been integrated into the Eclipse environment. As one of the aims of this project was to keep user input to a minimum, it was decided that many menus or buttons or options were not required, and only the bare minimum buttons were provided to trigger the analysis. Figure 4.7 shows the extra options (circled) that were added to the menu after right clicking on a package in Eclipse. Figure 4.8 shows the extra options (circled) that were added to the menu after right clicking on a class in Eclipse.

Similarly, if either a class, or a package is clicked in the Package Explorer view, a menu option is enabled allowing the user to perform the analysis, as shown in Figure 4.5. This minor addition to the user interface should increase usability, as developers have more than one way of invoking the analysis.

• Refactoring Scripting - The LTK needs refactoring scripts to execute refactorings. CoderAider contributes a refactoring contribution object which allows the refactoring framework to dynamically instantiate the CoderAider refactoring, and perform the changes specified by CoderAider.

4.3.1 CoderAider Components

In order to define a refactoring using the LTK framework, and to implement the refactoring lifecycle specified in Section 3.2.1, CoderAider specifies a number of components.

• Refactoring Contribution - This is a means of registering the CoderAider refactoring with the core refactoring framework, dynamically instantiating a refactoring object. This mechanism is used by the refactoring history service and the refactoring scripting service to recreate a particular, fully configured refactoring instance from a memento such as a refactoring script. This is a simple class, and the functionality is mostly inherited from abstract classes within the LTK.

• Refactoring Action - This is an action defined by CoderAider for interaction with the Eclipse workbench. A selection listener enables and disables the availability of CoderAider depending on whether a class or package or alternate element has been selected by the user. Once selected and chosen, this action invokes the CoderAider refactoring process (Step 1 of the lifecycle).

• Refactoring Wizard - This is used by CoderAider to present the refactorings in the user interface. This class provides the functionality to display the input pages, error pages and change preview panes depending upon the status of the change generation. Once again, the bulk of this functionality is made available to CoderAider by the LTK framework. We only need to specify the format of the input pages, and pass the relevant data onto the main Refactoring Class for processing by CoderAider (Steps 3 and 4 of the lifecycle).

• Refactoring Class - This is the principle component of the refactoring, and implements the main logic of CoderAider. This is the control class, and all actions carried out by CoderAider stem from here. The Refactoring Change object is generated by this class and passed to the LTK framework for integration into the Refactoring Wizard and Refactoring History. The way in which this object is generated is of most interest to us, and it is specified in detail in Chapters 5, 6, 7 and 8. (Step 4 of the lifecycle).
Figure 4.6: Undo

Figure 4.7: Right Clicking on a Package
4.4. CoderAider integration into Eclipse

Figure 4.8: Right Clicking on a Class
After the code transformation has been performed, it is integrated into the Eclipse Undo list, and therefore can be reversed if required, as shown in Figure 4.6.

Clicking on either a package or class refreshes the metrics view as well, as shown in Figure 4.9. This is a tab at the bottom of the screen. If a class is selected, it will only have one row, otherwise if a package is selected, it will have one row per class of that package. Each class has defined for it a number of metrics. The numerical value of each metric (rounded appropriately) is shown in each column. The meaning of each of these numbers is discussed in Chapter 9. This view is for the benefit of the developer to see how their code is in terms of quality. Internally, these metrics can be used for each class to improve the code analysis process by providing a greater certainty when deciding which parts of code need improvement.

Figure 4.10 shows the window that appears prior to the analysis commencing. It enables the user to choose which patterns to look for and analyse in the source code. These patterns are the refactoring specifications (code examples) provided by refactoring designers. They can come from existing projects, or other domain specific knowledge from designers, or even general design principles. The user can select which of these they feel necessary to apply to the current project. The analysis can then either be performed on the real source code by pressing OK, or a preview of changes can be seen by pressing Preview. This change preview is shown in Figure 4.11. It shows a graphical diff view of the old source code compared against the new source code - this allows developers to understand the contents of the changes, and decide whether or not they are appropriate. This view also allows developers to select which classes to apply the changes to (shown as a series of checkbox items in the top pane).
4.5 Summary

In this chapter we have looked at how CoderAider is integrated into a popular development environment - Eclipse. We have seen the features of the existing Eclipse code frameworks that have been used, and images of CoderAider in action. We have briefly covered the various components of CoderAider that were created in order to integrate with the Eclipse framework. We have also looked at how modifications are made to the source code.

The next chapter shall focus on one of the major contributions of this project - a methodology which enables developers to specify refactorings and good design practices with ease, possibly even by extracting this from existing projects.
Chapter 5

Refactoring Specification

“Each problem that I solved became a rule which served afterwards to solve other problems.”

- Rene Descartes

The core focus of this project was the creation of a refactoring system with a means of easily extending its capabilities. Before we describe in detail how this was achieved, it is useful to reflect upon some of the design decisions that were made, and alternatives that were considered. One thing which is immediately clear from the aims is that extensibility is vital. We should not hard-code any refactoring logic. All refactorings must be customisable, and it should be possible to add/remove new refactorings easily. Without this feature, the utility of this project would be severely limited. Extensibility ensures:

- Developers can share refactorings on the internet.
- Developers working within a certain domain (such as web applications for example) can share domain specific refactorings with each other.

Aside from extensibility, the other major challenge here is that of usability - existing refactoring tools are extensible, but writing new rules for them is a time consuming process, and needs careful studying of the documentation and syntax rules and so on. Ensuring that the specification methodology of CoderAider is highly intuitive is the key difficulty.

5.1 Desired expressibility

We need to consider what kinds of things it should be possible to express when writing a refactoring specification that CoderAider can use.

1. Find all classes that match certain properties. Transform them in a certain manner. Properties of a class may be:

   - Fields with certain visibilities, optionally final or static, having a certain type, and initializer. Need to be able to say whether it is essential that a certain property exists, or whether any value will do (e.g. we may not care about the visibility of a field, but we do care about whether it is static or not etc).

   - Similarly with methods, with the addition of parameters, and their types, quantity, and also method bodies matching a certain nature (perhaps an \texttt{if} statement which uses a field matched earlier).

   - The class inheritance hierarchy, and whether we want to match classes that inherit from certain classes only, or any classes at all.

Transformations on a class may be:
• Removal of a field that was described earlier.
• Removal of a method that was described earlier.
• Addition of a field.
• Addition of a method.
• Addition of a field initializer.
• Modification of field modifiers (e.g. final and static).
• Addition/removal of a method parameter.
• Transformation of a statement within the method body.
• Addition/removal of a superclass.

2. Find all classes that match certain properties. Delete them.

3. Add a new class consisting of certain properties. Use the method/field names of matched fields or methods as described earlier.

Most importantly, it should be possible to say any permutation of these things for any given source code example.

5.2 The XML Approach

The first idea for the specification of refactorings was by the use of XML. A prototype design was constructed to test the feasibility of this idea. For example,

```xml
<refactoring name = "Delete Field">
  <class name = "A" extends = "B">
    <field visibility = "public" type = "int" action = "delete">
    </field>
  </class>
</refactoring>
```

This refactoring is intended to search for any classes which extend B, and delete any attributes within them having properties public int. Similar XML constructs were used to specify actions to perform on methods and method parameters and so on. After some initial analysis, it was clear that this approach lacked flexibility. This was due to a number of factors:

• It was difficult to specify that methods should contain certain statements when being matched.

• Specifying things like “match this list of visibilities” became difficult, without leading to verbose XML.

• Differentiating between types and classes used in specifying the refactoring, and those types and classes in the given source code was not possible. For example, in the XML above, we do not know if A is a generic class used to name this refactoring (created by the refactoring designer), or if A is the name of an actual class that exists in the end-user’s code.

• Specifying how to change a field became difficult. Say we wanted to change its visibility to public and its return type to String. We would need to create another <field> node in the XML file and somehow link that to the existing <field> node and state that it is to be changed in the manner specified. This becomes quite complex to manage and design.

• Specifying that there may be a number of fields of a certain criteria is once again verbose. We may wish to state that there may be n fields which match a certain criterion, and they should all be modified in the manner specified. This sort of flexibility is not easily achieved.

In conclusion, we can see that while this XML approach will make analysis of refactoring specifications easier (as XML is easy to parse), it will make the definition of refactoring specifications a lot harder for refactoring developers. End-users will also be put off using the power of CoderAider for custom purposes if they have to write verbose XML to specify simple changes which should be made to code.
5.3 The Templates Approach

In most cases, it is obvious that both end-users and refactoring designers will be creating custom refactoring specifications based on their own notion of correctness, or desired result (behavioural or structural goal), and they will think of these notions in terms of code - i.e. they will think about the bad code they are trying to avoid and the good code they would like to convert it to. Alternatively, they will think about the existing code they have, and the code they would like to have instead. In fact, developers may think about code from previous projects, and may infer refactoring specifications based on lessons learnt in those projects - perhaps those projects contained elements of good design which should be applied to newer projects, or perhaps those projects contained poor design which should be avoided in the future. By tapping into this thought process, we can now explain the templates approach to refactoring specification. Indeed, it is this approach which CoderAider uses. The main benefits of this approach are simplicity, and the ability for developers to extract useful items of code from existing projects, and let them serve as (either good or bad) examples for newer projects. Therefore, this approach goes a long way towards crossing the hurdles presented when designing a specification system that is both extensible, and makes it easy for existing knowledge to be drawn upon.

At a very high level, the templates approach works as follows. In order to specify a refactoring, one needs to write out the code which is thought of as being “bad”, and the code which is thought of as being its improvement, i.e. the “good” code. This is known as a JavaCombinedTemplate. Alternatively, one needs to specify just the code which has some desirable structural properties (a JavaTemplate). The refactoring specification processor will extract from this specification the necessary transformations that should be made for the given project. These transformations can then be applied on the project by techniques described in Chapter 8.

Let us now consider how this approach works. First, we need to describe annotations of template source code.

5.3.1 Template Annotations

In order to add flexibility to the approach of writing example “bad” and “good” code for a refactoring specification, the template code needs to be annotated with some further information. Note that only the template code needs to be annotated - not the source code upon which it is being run. If these annotations are not present, in the template code, a straightforward match can still be performed, meaning that there is no unnecessary overhead for refactoring designers if they only wish to perform a simple code transformation. The implementation of these annotations is discussed in Section 5.3.5, but first we shall discuss the design.

Classes, methods, constructors and attributes in the template can all have annotations.

Classes

Classes can have the following annotations:

- Visibility - the list of acceptable visibilities when matching a class. The default is [Visibility.PUBLIC], but we can put any acceptable visibilities into the list.

- Extends - the class that must be extended by the current class being matched. The default is Extends.ANY, meaning that any superclass is acceptable.

  Alternatively, we can put Extends.SPECIFIED meaning that only the superclass written in the code of the template is acceptable. Alternatively we can have Extends.NONE meaning that the class should not have any superclasses.

- Implements - similar to Extends, but here we are interested in acceptable super interfaces. The default is Implements.ANY, meaning that any super interface is acceptable. Alternatively, we can put Implements.SPECIFIED meaning that only the super interface specified in the code of the template is acceptable. Similarly, Implements.NONE as well.
• Quantity - this is an important annotation. It specifies how many such classes we are expecting to perform the refactoring on. Contrary to what you might expect, this is not a number. It is usually not possible to say “perform this refactoring on only 3 classes”, and the usefulness of that statement is limited. Instead, we define quantity as being either Quantity.ONE (by default), or Quantity.MANY. Quantity.ONE means that only one such element (class in this case) is expected, and do not bother looking for any more after you have found that one element. Quantity.MANY means that many such elements are expected, so match all of them.

• Change - this is more of a convenience annotation, and a value of false means that only use this template for finding code which matches this template, but don’t transform that code into anything. We will see how this is useful later.

Attributes
Attributes can have the following annotations:

• Visibility - same as with classes.
• Quantity - same as with classes.
• Type - this specifies the type the attribute is expected to have. By default it is ReturnType.SPECIFIED, meaning that the attribute can only have the type specified in the template code. Alternatively, there is ReturnType.ANY, meaning that the attribute may have any type.
• Change - same as with classes.

Methods
Methods (and constructors) can have the following annotations:

• Visibility - same as with classes.
• Quantity - same as with classes.
• Return Type - same as with attributes. Ignored for constructors.
• Change - same as with classes.
• Parameters - by default this is set to Parameters.SPECIFIED, meaning that only the parameters specified in the template code are acceptable. We can also set this to Parameters.ANY_MORE, meaning that we must match the parameters specified in the template code, but it is acceptable if there are additional parameters too. There is no option to say that any method parameters are acceptable, as the match space for that would be too high - the template method would match too many methods in the real source code and making transformations would become harder.
• Uses - this is an optional string used to specify that a method must “use” a certain other class, type, field or method. How it “uses” it is irrelevant, a reference to that name is enough to say that the method “uses” that element.
• Comment - this is an optional string which will be inserted into any methods that match the code specified in the template. This can be useful for debugging purposes to find out which methods are being matched by a template method. More importantly, it is used when using Templates as Critics, as in Section 5.3.4.

As an example of annotations in action, consider the class in Listing 5.1. This can be used to find private classes that have an attribute in them which is final, has any return type, any visibility, and is initialized to null. This shows the flexibility of the annotation approach when writing code templates.
Listing 5.1: Example annotated class

```java
@ClassAnnotation ( extends = Extends.ANY, visibility = [Visibility.PRIVATE] )
class AClass {
    @FieldAnnotation ( visibility = {
        Visibility.DEFAULT, 
        Visibility.PROTECTED, 
        Visibility.PUBLIC, 
        Visibility.PRIVATE
    }, retType = ReturnType.ANY )
    final Object instance = null;
}
```

Listing 5.2: Example improvement template

```java
class Bad1 {
    @FieldAnnotation ( visibility = {
        Visibility.DEFAULT, 
        Visibility.PROTECTED, 
        Visibility.PUBLIC, 
        Visibility.PRIVATE
    }, retType = ReturnType.ANY )
    final Object instance = null;
}
class Good1 {
}
```

### 5.3.2 Improvement Templates

Using these annotations, we construct a “Bad” template (which contains example “bad” code), and a corresponding “Good” template (which contains its correction). The naming conventions for the template is what identifies which of the “Good” template classes are linked to the “Bad” template classes. e.g. The improvement for template class “Bad1” is given by class “Good1”, and so on. These templates are represented by `JavaCombinedTemplate` objects with CoderAider, as they consist of two separate templates - one for the bad code, and one for the good code.

In order to link the attributes between the “Bad” and “Good” templates, we ensure they have the same name. So, if the “Bad” template has an attribute of the form `public static final nameOfTheAttr;`, and the “Good” template has an attribute of the form `private static final nameOfTheAttr;`, then the refactoring specification processor will realise that any static final attributes that are public need to have their visibility changed to become private.

Similarly, we can link methods together as well by using the same name in both the “Bad” and “Good” classes.

Listing 5.2 illustrates the use of improvement templates to delete any final attributes which have been initialized to `null`. Note the deletion is implicit and realized by the processor due to the lack of the attribute named `instance` in the “Good1” class.
5.3.3 Design Pattern Templates

It is not essential to specify “bad” code, and its corresponding “good” code. There may be situations where even the level of flexibility provided by the annotations is not enough to cater for all possible variations of the “bad” code that may arise. In this case, it is simpler to write just some “good” template code. This is known as a JavaTemplate within CoderAider, as there is only one template. A good example of this is design patterns. We can create a template which contains a description of a design pattern, in code format. Annotations can be given to this as necessary. The refactoring specification processor will realise that this is code has some structure which is desired by the refactoring designer. It can then analyse the project source code for areas which are similar to this template, but differ slightly. These differences can then be converted to match this template, meaning that the project source code conforms to the template. More on this in Chapters 7 and 8.

5.3.4 Templates as Critics

Occasionally, situations will arise where the refactoring designer merely wants to highlight instances of a certain problem, but does not know how to go about solving that problem in an automatic fashion. We say that code of a certain nature needs to be criticized. In these cases, we use templates as what are known as critics. These critics will highlight (criticize) regions of code that have certain properties, but will not transform those regions. This highlighting can be done by insertion of a comment (hence the comment annotation noted earlier). This functionality is available at no additional cost (design or programming effort) when using the templates and annotations framework outlined in this section. It is merely a special way of using templates.

See Listing 5.3 for an example where templates need to be used as critics. Here we have a class A with a constructor which calls a method. Class B extends class A, and overrides that method. This overridden method uses a variable which hasn’t been initialized. Therefore, when constructing an instance of Class B, the overridden method will be called, and a null pointer exception will arise, meaning that instances of Class B cannot be created. This is an error, but it cannot be corrected automatically as it is not correct to delete calls to the overridden method - we simply cannot tell what the programmer intended to do here. So we need to criticize the code by inserting an appropriate comment.

5.3.5 Template Processing

The easiest way to enable processing of code annotation is by use of the Java APT. An annotation processor was created which firstly creates a model of the template source code (using techniques described in Section 6.4), and then augments the model with information harvested from the template annotations. If an annotation is missing, its default value is used for the augmentation phase. More details about augmentation are presented in Section 6.5.

The template processing stage is carried out every time CoderAider is invoked. This may seem like an additional cost overhead, and indeed it would be computationally more efficient if the template models were stored once generated. However, CoderAider permits refactoring designers to change templates and test them out immediately without further recompilation or restarting of the Eclipse workbench. This can only be achieved if templates are parsed and processed every time the project source code needs to be analysed. From a user perspective, the additional time this takes is minimal - one second at the very most, even on the slowest of hardware. It should be noted that this approach also makes the management of refactoring specification templates very straightforward. The refactoring designer needs only to insert / delete templates from the designated template folder for the templates to be recognised and used / discarded for analysis. It is for these reasons that the benefits brought by being able to see the effect of changes to templates immediately outweigh the computational inefficiency of having to process the templates on each analysis.

Note: Java APT - Annotation Processing Toolkit
Listing 5.3: Templates As Critics

@ClassAnnotation ( change = false , extend = Extends.ANY )
class A {
   @MethodAnnotation ( 
      change = false ,
      comment = ' ‘/CoderAider: Constructors shouldn’t call overridden methods’’
   )
   A() {
      method();
   }

   @MethodAnnotation ( 
      parameters = Parameters.ANYMORE ,
      returnType = ReturnType.ANY ,
      change = false
   )
   void method() {
   }
}

@ClassAnnotation ( change = false , extend = Extends.SPECIFIED )
class B extends A {
   @FieldAnnotation (change = false)
   private String somevar;

   @MethodAnnotation (change = false)
   B() {
      super();
   }

   @MethodAnnotation ( 
      parameters = Parameters.ANYMORE ,
      returnType = ReturnType.ANY ,
      change = false ,
      Uses = {"somevar"}
   )
   void method() {
   }
}
Chapter 5. Refactoring Specification

5.4 Summary

This chapter has divulged details of the unique refactoring specification methodology used by CoderAider, and how it may be made flexible. This includes using real code examples to specify good and bad practices when coding. We have studied an alternative approach, and found that the benefits of the suggested approach are:

- Easier to share refactoring specifications between developers that are working in the same application domain.
- Easier to extract refactoring specifications from existing projects.
- Easier to understand refactoring specifications.
- Easier to create new refactoring specifications.

This has achieved the main aims of extensibility and usability.

In Chapter 7, we shall focus on how to use these refactoring specifications to find areas of the user’s source code which match them - i.e. areas of the user’s source code that are potentially flawed and need some attention. Firstly, in the next chapter we shall look at how these refactoring specifications, and the user’s source code, can be represented internally for analysis purposes.
Chapter 6

Model Design

“Walking on water and developing software from a specification are easy if both are frozen.”

- Edward V. Berard

Now that a method of specifying refactorings has been described, we need to describe a model upon which these refactorings may be carried out. While the scope of this project is limited to performing refactoring upon Java code only, it was decided that this model should be made as generic as possible, to facilitate the future extension of CoderAider to cater for other languages as well. This involves the separation of object oriented elements from the Java-specific elements. This problem was solved by creating a generic model first which represented classes on a higher level, and a Java specific model which represented the Java classes and their relevant AST related logic for insertion, modification and deletion of nodes. This breakdown will be described shortly, but first we shall consider some design issues.

6.1 Design Issues

The challenge when designing a model is to decide what its intended purposes are, and ensure that the design can meet these requirements. Ensuring accessibility to developers means that the model should be easy to understand, create, and work with. We shall now discuss some impediments towards implementation that were encountered, and how these were overcome.

6.1.1 Custom Model or Existing Model?

Eclipse provides us with a framework for representing source code - we can obtain an AST. This AST is very detailed, and even figuring out which nodes of the AST correspond to our semantic notions of source code is difficult. Using this model entirely for this project will not be adequate, as too much time would be spent in dealing with the implementation issues surrounding this model such as how to represent a certain node, or how to modify a node, and so on. More importantly, this model does not conform to what we may expect for a code model representation. This model is, by definition, a tree. We will need to navigate this tree to find the information we need. For example, a class would be represented as a node, and its attributes and methods would be represented as its children. There would be no conceptual relationships, such as those that exist when one class contains numerous instances of another class, and so on. These conceptual links are of importance when creating a code analysis and improvement tool.

This leads us to our next option - using the UML metamodel of code. Unfortunately, this is very detailed, and contains several unnecessary elements that we do not require in this project. Such added complexity will introduce problems for us, as even conforming to its exact specifications may prove to be a challenge. Another difficulty would arise when converting the Eclipse provided AST (from the Eclipse workbench) into the UML metamodel. Due to the time constraints of this project, we need to ensure that such implementation issues do not hinder the main development work. Therefore, the decision was
taken to develop a custom model to represent code, and only provide the minimal elements required to successfully analyse code and perform transformations.

### 6.1.2 General Issues

After any model of the source code has been made, we believe that it is important that it is not cached. The reason for this is that the developer may have made changes to the source code since the last time the model was generated, and therefore caching is not appropriate. There is a speed vs utility trade-off here, as the model generation phase will take a long time on larger projects. However, we feel that this is the right decision as models which are cached are of little use in this project. It will also avoid unnecessary complexities such as lack of synchronicity between the user view of the code, and the internal representation of the code.

Another problem is deciding when analysis should commence, and when we should refuse to perform any analysis and transformation. The focus of CoderAider is that of code improvement using knowledge handed down from other developers, or from past projects. Analysing code which does not compile, or has syntax errors will prove to be problematic for CoderAider, as it means that at every step of the analysis and transformation of the original code, faults (parse errors) must be taken into consideration, thus adding an additional level of complexity. This also goes against the philosophy of CoderAider, which is concerned with improving the structural quality of the code in ways which cannot be suggested by the Java compiler or other existing tools. It is for this reason that we have added a precondition to the CoderAider analysis that the source code must compile correctly. If it does not, CoderAider will refuse to analyse it, and a model will not be built at all.

### 6.2 Generic Model

The first step involves specifying a generic model for code analysis. This model is generic enough to cater for any object oriented language, and also provides common functionality needed when developing any refactorings. The model used by CoderAider is specified in Figure 6.1. Some attributes and all methods have been removed for clarity. Classes can contain many Attributes, Methods and other Classes. Methods can contain Parameters. Note that no difference is made between interfaces and abstract classes here - a class will know which of the two it is, if any.

Now we look at the specification of classes, methods, attributes and parameters. Semantically, methods should contain many parameters, and attributes should not contain any parameters. Methods and attributes are unrelated otherwise. When designing a model, this semantic notion needs to be put aside, as there are similarities between methods, attributes and parameters which should not be ignored when creating a model. A Parameter is the most generic element, and it only has information about its name, its TypeDescriptor (see Section 6.2.1), and whether or not it is final. An attribute needs to store this information too, in addition to information about its visibility and whether it is static. Therefore, Attributes extend Parameters. Methods contain Parameters. However, in order to describe themselves, methods need to know the same pieces of information already contained in parameters and attributes, in addition to extras like whether it is a constructor and whether it is abstract. Therefore, Methods extend Attributes. Classes need to know all of the information that methods know as well (visibility, type, final, name, static and abstract). It is for this reason that Class extends Method. The aggregation in the diagram should be self explanatory.

#### 6.2.1 Type Descriptors

A TypeDescriptor is the CoderAider representation of the type of an element. This will hold information about the name of the type and the number of dimensions it has (0 for standard types, 1 for array types, 2 for 2D arrays and so on). A type is either the type of an attribute or method parameter, or the return type of a method. This representation is intended to make it easier to interact with the Eclipse AST which has a lot of detail in the type specification. This representation is made compatible with the AST
Figure 6.1: Class Diagram of the Model Package
representation by overriding the `equals` method to enable comparison of a `TypeDescriptor` with the Eclipse JDT types (e.g. `ArrayType`, `SimpleType`, `PrimitiveType` and so on).

There is also a static blank type, referred to as `ANY`. This is the type assigned to template attributes or methods where we do not care what the return type is, when matching source code. More on this in Chapter 5.

### 6.3 Java Specific Model

The next step in developing a model upon which analysis and transformation can be based was to specify a model that is specific to the the features of Java. Figure 6.2 shows how Java specifics can be added on top of the earlier defined generic model.

Note the extra `JavaModel` class now. This is the key element of this model. This class stores the model of the given source code. It is generated by using a model generation process, described in Section 6.4. It stores a list of `JavaClasses` which represent the source code being analysed. It also stores the JDT specific `CompilationUnit` and `ASTRewrite` objects. The importance of `ASTRewrite` was discussed earlier in Section 4.2.1. In essence, it is used to store a list of transformations that will be made to the code. Therefore, every `JavaClass`, `JavaAttribute` and `JavaMethod` needs to be passed this object while being constructed. They will store this object, and use it to record any modifications on (such as visibilities being changed and so on). `JavaModel` has an important method called `applyChanges`, which writes out the changes specified by the `ASTRewrite` object onto the source code. This method will be invoked by the LTK framework once the user accepts the proposed changes (see Section 3.2.1, Step 5). For implementation efficiency, the `JavaModel` class also stores two maps of `JavaClass` to a list of methods.
and attributes in that class.

The JavaAttribute class has an extra field to store the initializer of the attribute, if any. This is stored as a VariableDeclarationFragment. In fact, here we slightly deviate from the Java syntax, which states that each attribute has stored with it a list of VariableDeclarationFragments. This is because Java lets developers define attributes like this:

```java
int number = 3, k = 0;
```

Storing that in a model is quite cumbersome, so CoderAider will recognise that as being the same as:

```java
int number = 3;
int k = 0;
```

In this way, it will be stored as two attributes, each with single a VariableDeclarationFragment. This makes analysis simpler and removes the unnecessary complexity which the Java syntax brings. Note that thinking of this as two separate attributes does not mean that the transformed AST will contain two separate attributes. The AST rewriter will automatically take care of any modifications to the attributes and leave the style used by the programmer intact.

### 6.4 Model Generation

Now that we have decided upon a model to represent any given source code, we must outline a method of generating this model. In the implementation, it is the JavaModelGenerator class which generates models. This model generation phase is carried out every time CoderAider is invoked for analysis, as described earlier when discussing design issues.

1. The model generator parses the compilation unit passed into it, using the ASTParser provided by the JDT, which returns an AST. The ASTParser will still return an AST if the source code passed into it does not parse or compile correctly, however not all of the AST will be correct (as is to be expected). As discussed previously, we will ignore the case when there are compilation errors (and refuse to analyse further).

2. The result of parsing the source code is a compilation unit. This is used to construct the model generator. After this, the generate method is invoked on the model generator. The JDT provides a Visitor pattern for traversing every node of the AST of the source code. The model generation method takes advantage of this by asking the compilation unit to accept itself. Note that the model generator is actually an ASTVisitor.

3. The model generator has visit methods for MethodDeclarations, FieldDeclarations, and TypeDeclarations. This is used to construct the different Java model elements, as per the design outlined earlier in this chapter.

4. A JavaModel (see Figure 6.2) is generated by this generator, and this can be retrieved by clients at any time for further processing. CoderAider code analysis is one such client of this model generation, and can use the model generated for analysis and transformation purposes. Note that this JavaModel only contains a list of classes which clients can retrieve, and an ASTRewrite object, which clients may use to schedule changes to the code.

Note that this model of the source code which has been generated has no relevance to any refactoring aspects of the project. It is merely a basis upon which the refactoring logic can be built. This leaves it open to the possibility of being used by other plugins for other purposes as well. For further information about one such example of this, see Chapter 9.
6.5 Model of a template

Chapter 5 introduced the concept of a source code template. This template can be processed in the same way that normal code is processed. The `JavaModelGenerator` class is used to construct a model of the template in a manner described earlier. This is followed by an augmentation phase which inserts the values of the annotations of classes, attributes and methods in special “template” attributes in the generated model. Therefore, annotations for classes will be stored in “template” attributes in `JavaClass`, and similarly for `JavaAttribute` and `JavaMethod`. These objects are also marked as being template objects. For example, if an attribute had an annotation of `Quantity.MANY`, then there would exist an attribute within its corresponding `JavaAttribute` representation, which would hold this information. Therefore, whenever this `JavaAttribute` is accessed, we can find the values of its annotations immediately, and this is very useful when performing analysis or transformation on code.

This method is the most efficient method of processing templates because it leverages the existing model creation infrastructure. This is another reason why the templates approach proved to be the most efficient approach for the specification of refactorings.

6.6 Summary

In this chapter, we have discussed potential methods of representing the source code, and have outlined the strategy that will be used for code representation within CoderAider. We have also highlighted its benefits and limitations, and shown how it may be generated from a practical perspective.

In the next chapter, we shall look at how an internal representation of the code can be used to perform matching - i.e. how we can match regions of the user’s code against the refactoring specifications to find areas in need of improvement. In Chapter 8, we shall look at how the model of the user’s code can be updated in order to add necessary code improvements.
Chapter 7

Matching of Patterns within Code

“It is unworthy of excellent men to lose hours like slaves in the labour of calculation, which could be relegated to anyone else if machines were used.”

- Gottfried von Leibnitz

After the specification of refactorings, and the design of a model for the internal representation of code, the next logical step is the application of the refactoring specification onto the code representation. This can be broken up into two areas:

- Match patterns - find regions of the given source code which need to be considered, according to the refactoring specification
- Transform code - transform the found regions of source code in the manner stated by the refactoring specification

This chapter deals with the first of these two items, and Chapter 8 deals with the second. The main challenge to be overcome here is how we can find a match, even if that match is a variant of the pattern that we are looking for. Overcoming this difficulty is the main aim of this chapter. Obviously it would not be of much benefit to anybody if the matching process only found identical matches - this would mean that far too many code examples have to be written to be of any use, and this is not practical. But matching variants opens up a whole new problem - false positives. We may think that a variant is actually a match when it is not. Therefore the key challenge is to achieve an appropriate equilibrium point between being able to match variants, and not generating too many erroneous results.

One option we may look at is using code similarity algorithms as used in duplicate code detection. Unfortunately, these have the key disadvantage that they are designed for use on code itself, rather than over structures such as classes. We are more interested in finding classes that match certain templates and so on.

Another option is using the Eclipse JDT - this provides an ASTMatcher class which is a visitor that can be used to search for matches in a given AST. This may seem ideal for the purpose of matching template source code to the user source code, but it has some severe limitations. It is not possible to define a generic notion of template types, nor is it possible to attach a weighting to the importance of matching different elements. This matching technique is better suited to matching simple pieces of code, as it performs a structural match, but only returns a boolean value rather than a numerical match certainty. Therefore, we need to create our own matching algorithm. In fact, the one we have created is a hybrid of our own logic and the Eclipse JDT functionality provided by the ASTMatcher.

Before we proceed, a quick word on terminology used. “User classes”, “user methods” and “user attributes” refer to the elements of the source code which are being analysed by the user. “Template classes”, “template methods” and “template attributes” refer to the refactoring specification templates defined in Chapter 5.
Figure 7.1: Elements of class analysed in the matching phase

Listing 7.1: Matching algorithm

```java
for each template T {
    for each class C in template T {
        for each class U in the user’s given source code {
            if (match(C, U) > 80%) {
                C is a match for U in T
            }
        }
    }
}
```

7.1 Classes

Firstly, we need to consider how to match a template class against the given source code - i.e. find classes in the given source code which have properties defined by the template class. This can be broken down into several areas such as considering the superclasses and superinterfaces of the class and considering its attributes and methods. See Figure 7.1 for a detailed breakdown of a few of the things that we need to analyse when matching a class.

This section will describe each of these in detail, and some complications that arise will be explained in later sections. The general approach for doing this matching is shown in Listing 7.1. This section describes the match function which takes a template class and a class from the source code model and calculates a percentage of the template class properties that are satisfied by the class in question. This is not a true percentage - it is more like a likelihood estimate, calculated using a weighted scoring procedure. The procedure is adjusted to give a result of 100% only if every template property matches.

Every element that is considered during the matching phase is given a score. At the end, a weighted sum is taken of the scores. This is because it is thought that matching some elements is more important than matching other elements. Obviously, this weighting is subject to debate. The option we suggest is
7.1. Classes

Listing 7.2: Super class matching

```java
@ClassAnnotation(
    extends = Extends.SPECIFIED
)
class A extends B {
    ...
}

class B {
    ...
}
```

as follows, and was obtained after performing some trial runs to judge performance:

\[
\text{score} = 2 \times \text{ExtendsScore} + 3 \times \text{AverageAttributesScore} + 3 \times \text{AverageMethodsScore} \tag{7.1}
\]

At the moment, this logic is encoded into the code, but it will not be difficult as part of a future extension to have this as custom annotations for various elements in the refactoring specifications, so that designers can assign weightings to the various elements themselves. The way that each Extends Score, Attribute Score and Method Score is calculated will now be described.

7.1.1 Super classes

If a template class has annotation of Extends.ANY, then the Extends Score is set to 1, and no comparisons are made - every class satisfies this condition. If the template class has an annotation of Extends.SPECIFIED this means that a comparison needs to be made between the class that is specified as the superclass in the template class, and the superclass of the class in the source code. There are three possibilities:

1. The superclass specified in the template class is another template class. We can find this out by looking to see if there are other template classes of the same name. If it is, then we add that template class to the list of dependencies of the current class. In other words, we need to first find a list of classes in the source code that match the template class named as the superclass. We can then compare to see if one of that list is the superclass of the class being compared against. e.g. Say we have a template like in Listing 7.2, and we are finding matches for template class A, and we are considering whether class U from the user’s source code is a match. First we need to find matches for template class B. Then, we see if one of those matches is a superclass for class U. If it is, then set the Extends Score to 1. However, we cannot set this score straight away until template class B has been analysed for matches. Therefore we add this as a “TODO” (see Section 7.2), i.e. the score will be updated at a later stage.

2. The superclass specified in the template class is not a template class (i.e. there is no template class of the same name). In this case, we can do a direct comparison to see whether the class U (being considered as a potential match from the user’s source code) has that named superclass.

3. There is a superclass specified in the template class, but class U has no superclass (or vice versa). In this case, we do not care whether the superclass specified in the template class is another template class or not. We merely set the Extends Score to 0, as the condition is not satisfied.

The above argument is valid for superinterfaces as well, where we consider the Implements.ANY or Implements.SPECIFIED annotations instead. In fact, the above argument is used throughout the matching process whenever a type, method or attribute is encountered that refers to a template type, attribute or method.

7.1.2 Attributes

For every attribute listed in the template class, we consider whether there is a matching attribute in the class U. For each of the attribute modifiers, visibility, type and initializer, we increment the Attribute Score by 1 if it matches the template attribute. Finally, we divide the Attribute Score by 4, as there
Chapter 7. Matching of Patterns within Code

Listing 7.3: Attribute matching

```java
for attribute A in template class C {
    for attribute UA in user class U {
        if (match(A, UA) > 80%) {
            A is a match for UA
        }
    }
}
```

are 4 things being considered when matching attributes. To summarize, Listing 7.3 shows the matching process for attributes.

There is one additional issue worthy of mention. The Quantity annotation of attributes tells us how many such attributes we can expect to find. If it is set to Quantity.ONE (default), we can stop looking for more matches after we find one match. This match is stored in a hashmap which maps the names of the template attributes to the names of their matching counterparts in the user class U. If the annotation was set to Quantity.MANY, we can expect to find more matches after one match is found. Therefore, we keep looking, and add every match found to the hashmap. In this case, looking up the name of the template attribute in the hashmap will yield a list of matching attributes from the user class U. This will later assist us when making transformations.

Attribute type

If attribute A in the template class does not have the same type as attribute UA in the user class, it cannot match. If we allowed this to match, there would be too many false positives - almost any two attributes will be thought of as matches. Therefore the type must match, otherwise we set the score to 0.

To see if the type matches, we follow the same procedure that we followed when considering the superclasses. If the type is a template type, then we add that template to the dependency list. We need to see if one of the matches for that template is the type for this attribute. Otherwise we do a straightforward comparison.

Attribute modifiers

The attribute modifiers include the visibility modifiers, and keywords like final and static. For visibility, we consider the list of acceptable visibilities specified in the template attribute annotations. If the attribute in question has one of these visibilities, we can increment the score. Similarly for other attribute modifiers too.

Attribute initializers

Here, we compare whether the initializer of the template attribute is the same as the initializer of the user’s class attribute. In order to do this, we take advantage of the ASTMatcher. We define our own matching class which extends the ASTMatcher, but also has a reference to the list of classes matched by every template class, and the hashmap that maps the names of the template attributes to the names of their user-class counterparts. Then the matcher visits every node in the initializer as normal, but if it ever encounters a name (of a template class, or a template attribute), it will substitute it with the name of its matched counterparts before deciding whether or not to accept it as a match. Say we have a set of attributes in the template class as shown in Listing 7.4. In order to find matches for anotherField in the user class, we need to first find matches for aField. We can then use the ASTMatcher variant to do a structural comparison on the initializer of a potential user attribute to see whether it matches aField + 1. When we come to compare the aField part of the structure, obviously it won’t match directly because
7.1. Classes

Listing 7.4: Attribute matching

```java
int aField = 0;
int anotherField = aField + 1;
```

Listing 7.5: Method matching

```java
for method M in template class C {
    for method UM in user class U {
        if (match(A, UM) > 80%) {
            M is a match for UM
        }
    }
}
```

`aField` is a template attribute. Therefore, we consult the hashmap to find its corresponding matches, and see whether one of those is used in the initializer for the user attribute being considered. Therefore a set of attributes like “int userField = 0; int anotherUserField = userField + 1” will be matched, but “int userField = 0; int anotherUserField = something + 1” will not be matched.

7.1.3 Methods

The treatment for matching methods is similar to that of attributes, so details will be kept to a minimal. For every method in the template class, we attempt to find a matching method in the user class. Scoring for the match is identical, but different items are considered. Listing 7.5 shows the details of matching template methods against those in user classes. The `Quantity` annotation is applicable to methods too, and is treated in the same manner as described for attributes above.

Method body

The matching of template method bodies against the bodies of user methods is similar to that of matching the attribute initializers. A class that extends the JDT `ASTMatcher` is used to perform this matching, but this time, a partial match is enough. In other words, the user method body must contain all of the statements in the template method body, but is allowed to contain more too (those will be ignored, but the position where the template code matched will be stored for later consideration). This visitor class proceeds in a similar fashion to that of matching attribute initializers - it has information about which template classes have already been mapped, and which attribute names and method names have been mapped to user attributes and methods, and therefore can use this information to perform substitutions when doing a comparison of equality.

Method parameters

The matching of method parameters depends on the method annotation for parameters. If it is set to `Parameters.SPECIFIED`, then only user methods, that have all matching parameters as the template method, will match. If the annotation is set to `Parameters.ANY_MORE`, then the user methods must contain all of the parameters (in the same order) that the template method had, but it is allowed to contain further parameters too. Parameters themselves are thought of as a special case of attributes, but with no initializer, and the matching process is identical.
Method modifiers

Once again, annotations such as Visibility are taken into account when comparing the visibility, as that defines the list of acceptable method visibilities. If the keywords static, final or synchronized are present in the template method modifiers, then the matcher will check for their presence in the user method being considered as a potential match.

Method Return Types

If the method has an annotation of ReturnType.ANY, then no further matching is done on the return type - any user method satisfies this condition. If the method has an annotation of ReturnType.SPECIFIED, then we must check to see if the return type specified in the template method declaration is the same as the return type of the user method under consideration. We have already solved this problem, and this equality check follows the same rules as with the check for matching method superclasses described in detail earlier.

7.2 Dependency Resolution and 2-phase matching

Throughout the previous section, we have implicitly referred to these things, but it is worth considering them explicitly. The matching process is carried out in two phases. The first phase matches everything that is straightforward, and keeps a dependency list for items which need to be matched, but cannot be matched just yet. This happens when for example there is a method return type which is actually a template type - the template class in question needs to be matched first. The second phase of the matching is the same as the first phase, but this time we have more information to perform the match, and the matching progresses in a special order. Listing 7.6 outlines the processes at work here, in pseudocode.

After the dependencies are collected in the first phase, the matches are sorted in dependency order for the second phase of the match. Consider the following example where A-E are classes:

- A depends on B, C
- B depends on C, E
- C depends on nothing
- D depends on A
- E depends on nothing

So matching in the second phase must be performed in this order: C, E, B, A, D. This means that when matching class B for example, the matcher knows which classes are matches for classes C and E, and therefore whenever a reference to C or E is encountered, it knows which classes are acceptable there (i.e. those that have already been known to match classes C and E).

This dependency analysis must proceed with caution. We must detect loops in the dependency graph to avoid entering the case where matches cannot be ordered for processing. If a loop is encountered, it will be broken, and less information will be made available to the matcher. Similarly, if a class is dependent on itself, then we proceed without looking up the list of matched classes. e.g. Say we have a template like Listing 7.7. This template class is dependent on itself because an attribute contains a template type A. In order to decide whether a given attribute of a user class matches the attribute instance in the template class, we need to consider the type of the user attribute and whether it matches one of the user classes that match template class A. This is treated as a special case, and we compare the type of the user attribute against the name of the user class in which it is contained. A similar procedure is followed for other cases where a class refers to itself (in method return types, parameters, bodies etc). The rest of the matching for the class will proceed as normal (i.e. the template attribute kAttr and the constructor will be matched as normal).
7.2. Dependency Resolution and 2-phase matching

Listing 7.6: 2 phase matching

// First phase
for each Template in List<Template Class> {
    for each Class in List<User Class> {
        match (templateClass, userClass);
        if (match is likely) {
            add to MatchSet (match);
        }
    }
}

// Sort
for each match in MatchSet {
    if (match has 0 dependencies OR
        match is dependent on itself) {
        // We will analyse it first
        add to beginning of SortedMatchList;
    }
}

while (SortedMatchList doesn't have all items in MatchSet) {
    for each match in MatchSet {
        if (SortedMatchList doesn't have match AND
            SortedMatchList has all of the dependencies of match) {
            add match to SortedMatchList at the end;
        }
    }
}

// Second phase
for each match in SortedMatchList {
    // All required dependencies have been resolved to analyse this match
    analyseMatch(match);
}

Listing 7.7: Class dependent on itself

class A {
    private A instance;
    private int kAttr;
    public A() { ... }
}
7.3 Properties of the matching process

For the algorithm proposed in this chapter, we can create an estimate of its complexity. For each template class, we shall perform one pass through all of the user classes for the first phase of the algorithm. Then we shall sort the matches found in dependency order, and the maximum number of loop iterations this can take is equal to the number of user classes squared (in the worst case scenario). After this, we must perform the second phase of the algorithm, which can involve a worst case scenario of performing as many loop iterations as there are user classes. To summarise, this gives us a complexity of $U^2 + 2U$ where $U$ is the number of user classes. This does highlight the limitation of this algorithm that it will get progressively slower with larger codebases to analyse.

Termination is another property which we must ensure. Developers will not use CoderAider if it cannot guarantee termination. In fact, termination is guaranteed by this algorithm, as we are performing analyses of each class in a serial manner. The sorting of the classes by dependency order also ensures that we do not get stuck in a loop, and we have the necessary information to proceed in the second phase of the algorithm.

This algorithm is also idempotent - if it is run for a second time, it will find exactly the same matches as the first time. This is a useful property, as it means developers can analyse their code any number of times for a match, and the same results will be guaranteed each time.

7.4 Result of the matching process

The final result of the matching process is a map that, for each template class in a template, lists the user classes that match it. For each match, there is also a map which maps attribute and method names in the template to those matched in the user class. This will help the transformer to realise why a user class matched a given template class, and will help identify the points where transformations need to be made. Figure 7.2 shows a graphical view of this map.

This matching procedure can match any user source code which is similar to the example code provided.
in the refactoring specifications. In other words, if you can write some example code, then you can also find similar code in the user’s codebase. Note that we are mainly referring to example code structures (such as classes with attributes and methods) rather than detailed statements in method bodies, although these can be matched using this procedure as well. Unfortunately, it is difficult to write refactoring specifications which will be guaranteed to only match certain code, and not other code. Due to this, now and again, we find too many matches for a given refactoring specification, meaning that the specification needs to be rewritten more carefully, or that more annotations need to be introduced in order to limit its matching capabilities. Another limitation is that if for any reason you can’t write any example code to show what kinds of things to look for in the user’s codebase, then you can’t use this matching procedure.

7.5 Summary

This chapter has described a novel matching procedure which can be used to determine regions of the user’s source code that match certain examples provided by refactoring specification designers. These regions will be the regions which need attention, as they may be flawed. We have also shown how code can still be matched (against a code example from a refactoring specification) even if it is not identical to it, and has some structural variations. This will prove useful when satisfying the aim of ensuring one code example in a refactoring specification can describe a whole class of refactorings without having to monotonically specify each and every possible combination or variation.

Now that we have identified which regions of the user’s code match the “bad” code examples provided in the refactoring specifications, the next step is calculating the differences between the “bad” and “good” code examples provided, and applying the same transformations onto the user’s source code. This is the focus of the next chapter.
Chapter 8

Code Transformations

“Get your facts first, then you can distort them as you please.”
- Mark Twain

The previous chapter described how given some refactoring specifications, we could find areas of the
user’s source code where those refactorings could be carried out. Now that we know where the refactorings
are supposed to happen, we have to describe a procedure for making the appropriate transformations.
That is the core focus of this chapter.

After the matching phase, an object (MatchSet) was produced which:

- For each template, maps each template class to a class in the user’s project
- For each template attribute in that template class, maps it to a list of attributes in the user’s class
- For each template method in that template class, maps it to a list of methods in the user’s class

To recap, there are two kinds of template. The first is a JavaCombinedTemplate which consists of a
template of “Bad” and “Good” code, showing how a certain type of problem may be fixed. The second is
a JavaTemplate which is a representation of code having some desired structural properties (e.g. a design
pattern). The processing of these is very similar, as we always search for matches to the template, and
then seek ways of transforming the matched elements. In the case of templates representing bad coding
practices, this is simple as once we have matched that same problem in the user’s source code, we can
use the good template to figure out how to correct that problem in the user’s source code. In the case of
templates representing code with desired structural properties, once we find a match, we can transform
it to become more of a match, e.g. the template of a desired code structure may suggest a certain method,
but the user’s source code may be missing that method, so we can add it.

The main challenge is to figure out what changes we need to make to the user’s source code. To do
this, we should work out the differences between the “bad” and “good” code examples, as noted above.
Obviously, this is not enough to figure out what changes to make to the user’s source code, and this is
where the challenge lies - the user will have used different variable, class and method names than the
example templates, so if our templates tell us to change the visibility of a method A, then we will actually
need to change the visibility of the method corresponding to A in the user’s source code.

Another major challenge is that of finding the sequence in which to apply transformations. In the
approach we have selected, we are merely using code examples to determine when changes are needed,
and which regions they are needed for. This means that it does not matter which order we apply the
templates to the user’s source code! Within each template, a sequence of steps will be found to convert the
“bad” code into the “good” code. The transformation is only complete if all of these steps are completed.
Therefore, once again, it does not matter which order we perform these steps. This illustrates the key
benefit of our approach, and many existing tools struggle to decide the ordering of refactorings.

Overcoming the hurdles of behaviour preservation deserves a mention here. CoderAider does not
concern itself to any major degree about whether behaviour of the user’s source code is being preserved,
or improved for the better. It assumes that the code examples of good and bad code provided in
the refactoring specification were created by a knowledgeable developer who knows the implications of
transforming code from the “bad” version to the “good” version. Therefore, it is safe to ignore this
aspect.

We can now describe the algorithm behind transformations. We proceed as follows. Note \( \eta \) is an
integer \( \geq 1 \). Repeat for every \( \eta \) in every template \( \Gamma \):

1. Consider class \( \text{Bad}_\eta \) of \( \Gamma \), where \( \eta \geq 1 \).
2. Now find class \( \text{Good}_\eta \) (in \( \Gamma \)) that represents the good code equivalent.
3. Lookup in the MatchSet the set \( \Upsilon \) of classes of the user’s source code that correspond to \( \text{Bad}_\eta \).
4. For each class \( \upsilon \) in the set \( \Upsilon \), transform it to match \( \text{Good}_\eta \). Details in Section 8.1.

If \( \text{Bad}_\eta \) exists in \( \Gamma \), but \( \text{Good}_\eta \) does not exist in \( \Gamma \), we must delete every class in the set \( \Upsilon \), as the refactoring
specification states that the way to improve those “bad” classes is by removing them. Similarly if \( \text{Good}_\eta \)
exists in \( \Gamma \), but \( \text{Bad}_\eta \) does not exist in \( \Gamma \), we must add a new class to the user’s project which matches the
\( \text{Good}_\eta \) class.

### 8.1 Transformation of Classes

We are now concerned with transforming the user class \( \upsilon \) to match the template class \( \text{Good}_\eta \) in the
template \( \Gamma \). We proceed as follows.

1. For every attribute \( \delta \) of \( \text{Bad}_\eta \), find the set \( \Theta \) of corresponding attributes in \( \upsilon \).
2. For every attribute \( \theta \) of \( \Theta \), transform it to match the corresponding attribute for \( \delta \) in \( \text{Good}_\eta \) (\( \alpha \)).
   Note that \( \delta \) and \( \alpha \) will have the same name. Details of the transformation in Section 8.1.1.
3. For every method \( \kappa \) of \( \text{Bad}_\eta \), find the set \( \Omega \) of corresponding methods in \( \upsilon \).
4. For every method \( \omega \) of \( \Omega \), transform it to match the corresponding method for \( \kappa \) in \( \text{Good}_\eta \) (\( \zeta \)). Note
   that \( \kappa \) and \( \zeta \) will have the same name. Details of the transformation in Section 8.1.2.
5. Ensure that the superclasses/superinterfaces of \( \upsilon \) match those specified by \( \text{Good}_\eta \). Details in Sec-
tion 8.1.3.

#### 8.1.1 Transformation of Attributes

We are now concerned with transforming the user attribute \( \theta \) to match the attribute \( \alpha \) in the template
class \( \text{Good}_\eta \).

- If there is no such \( \theta \), we must clone the template attribute \( \alpha \) and add it to the user class \( \upsilon \).
- If there is no such \( \alpha \), we must delete the attribute \( \theta \), as it means an attribute exists in the template
class \( \text{Bad}_\eta \), but a corresponding attribute (of the same name) does not exist in the template class
\( \text{Good}_\eta \).
- If \( \alpha \) has an annotation of \text{change} = \text{false}, then we don’t want to perform any changes on the user
attribute \( \theta \).
- Otherwise, we proceed as follows.
  - We ensure the visibilities and modifiers of the two attributes \( \theta \) and \( \alpha \) match. If not, we use the
reference to the \text{ASTRewrite} object to schedule a change for attribute \( \theta \) to change the required
visibility and/or modifiers.
8.1. Transformation of Classes

Similarly we compare the initializers and the attribute types, and schedule rewrites as necessary. If the attribute type is a template type (we know this because it will be of the form Goodγ), then we ensure that θ has an attribute type which corresponds to one of the items in the list obtained by looking up Goodγ in the MatchSet.

8.1.2 Transformation of Methods

We are now concerned with transforming the user method ω to match the method ζ in the template class Goodη.

- If there is no such ω, we must clone the template method ω and add it to the user class υ.
- If there is no such ζ, we must delete the user method ω.
- If ζ has an annotation of change = false, then we don’t want to perform any changes on the user method ω.
- If ζ has a comment annotation, we insert that comment at the top of the user method ω. This is done by using the reference to the ASTRewrite object contained in the method ω.
- Otherwise we proceed as follows.
  - We ensure the visibilities and modifiers of the two methods ω and ζ match. If not, we use the reference to the ASTRewrite object to schedule a change for method ω to change the required visibility and/or modifiers.
  - Similarly, we ensure that each parameter of method ζ is a parameter for method ω (parameters can be thought of and processed in the same way as attributes, ignoring the missing visibility of parameters).
  - Ensure that the return type for the method ζ matches the return type for the user method ω. If it does not, we need to consider what happened. ω had previously matched κ. This means that the return type of method κ (of Badη) did not match the return type of method ζ (of Goodη), implying that a change of return types was intended. We must find the appropriate return type to use for ω, by using techniques already described for transforming the object types of attributes.
  - We also need to ensure that the body of the method ζ is contained in the user method ω. We can do this by using a custom ASTMatcher in a manner similar to that described in Section 7.1.3. If it is not, then we need to add the statement(s) in the body of ζ to the body of ω in the appropriate position. This position can be calculated by looking for similar statements in ω, otherwise it will be inserted at the top and flagged for user approval. The statement(s) can only be inserted straight away if they contain no references to template classes or types. Otherwise, we must use a StatementRenamer. This is a visitor that will visit every node of the statement(s), and schedule appropriate renaming of template types using the ASTRewrite object and the MatchSet to lookup the appropriate values for the template names in use. There is an additional complexity worth noting here. If the template type in question is actually referring to an attribute with an annotation of Quantity.MANY, it means it is referring to a set of possible values. In order to determine which of these values to map it to, we must look at the user method ω to see which value is appropriate.

8.1.3 Transformation of Superclasses

We are now concerned with transforming the superclasses and superinterfaces of the user class υ to match those of the template class Goodη (only if annotation of Extents.SPECIFIED is present). For the case of superclasses, we firstly remove the existing superclass of υ, and add the superclass specified by Goodη. If the superclass specified by Goodη is a template class (we know this because it will be of the form Goodγ),
then we insert the corresponding user class obtained by looking up $\text{Good}_\gamma$ in the $\text{MatchSet}$ obtained from the match phase.

If $\text{Good}_\eta$ lists a superclass or superinterface, but $\text{Bad}_\eta$ does not, then an appropriate superclass or superinterface must be added. By appropriate, we mean check if it is a template class or not, and perform the mapping as described before. Alternatively, if $\text{Good}_\eta$ does not list a superclass or superinterface, but $\text{Bad}_\eta$ does list a superclass or superinterface, then we must remove that superclass or superinterface from the user class $\upsilon$.

### 8.2 Transformations Wrapper

Throughout the previous section, we referred to using the $\text{ASTRewrite}$ object to change visibilities and modifiers of both attributes and methods. This is a useful method for scheduling changes without actually making them, and it merits have already been considered in Section 4.2.1.

Unfortunately there are a few pitfalls to be noted. The $\text{ASTRewrite}$ is a very simple rewrite scheduler, and it does not perform any sort of checking to ensure consistency or integrity. For example, if multiple refactoring specifications modify the same attribute’s visibility to private, the word private will appear multiple times in the attribute definition in the rewritten source code, leading to a compilation error. To stop this from happening, we have created a wrapper for inserting changes into $\text{ASTRewrite}$. This wrapper will consider which changes are already scheduled to be made to the property under consideration (e.g. the visibility of an attribute), and will decide whether or not to schedule the requested change.

Another function of the wrapper is to make it easier to specify changes. If we wished to change an attribute visibility to private, the $\text{ASTRewrite}$ object supplied by the JDT would only add the private keyword to the attribute. We must schedule additional changes - namely the removal of all other visibility modifiers from that attribute (e.g. public and protected). The same applies for other modifiers such as static and final.

### 8.3 Summary

Chapter 7 showed how we can use the “bad” code examples in the refactoring specifications to find regions of similar code in the user’s codebase. This chapter has shown how we can:

- Find the differences between the “bad” and “good” code examples in the refactoring specifications, and the transformations needed to convert the “bad” example into the “good” example.

- Apply the same transformations onto the user’s source code, in order to make it comply with the “good” code example.

The following chapter discusses the use of metrics to further improve this process, as well as highlight how the code model designed for CoderAider can be utilised for alternate purposes, showing its flexibility.
Chapter 9

Metrics Framework

“Anyone who considers arithmetical methods of producing random numbers is, of course, in a state of sin.”

- John von Neumann

Quantitative approaches using metrics can assist developers in detecting potential design flaws and finding useful restructurings to correct them [34]. Indeed, if such restructurings could be automated with a quantitative and formal basis, design quality and development productivity could be improved [21] as well as monitored and compared with other designs.

In the case of CoderAider, we have already outlined refactoring specifications, and a matching and code transformation procedure. We can make use of metrics to aid the process of finding matches when it is unclear whether the user’s code is similar to the given code example in the refactoring specification. We can also make use of metrics to determine whether the transformation being suggested in the refactoring specification will actually be of benefit to the user’s codebase (i.e. whether it is applicable in this case or not). The challenge here is how best to utilise metrics to do this. This is difficult because a metric may be more effective for one project than another. In fact, one metric may be more beneficial to a whole class of applications than another class, in other words it may be domain specific. This means that CoderAider needs to be extensible enough to cater for developers wishing to define their own metric. These defined metrics may then be shared with other developers, and built into CoderAider for future releases.

Chapter 6 discussed the specification and generation of a custom model upon which operations can be carried out. The main operations illustrated by this project involve the analysis and transformation of code in order to improve its structure from a design perspective. This model can also be used to define a metrics framework, which as discussed above, may be beneficial to improve the matching and transformation processes in the previous two chapters. The GUI of the designed metrics framework will display a tabular form of different metrics for the source code, based on the model generated. Figure 4.9 shows the final result.

9.1 Architecture

Figure 9.1 shows the architecture of the metrics framework. A metric is defined as an extension of the AbstractMetric class. This class contains a reference to the JavaClass for which this metric is based, and contains a set of JavaModels which are the representation of the project being analysed. This class also provides common functionality required by any metric, such as finding the superclass of a class, finding children of a class and so on.

This abstract class defines two methods which all children must implement. One is the getValue method which returns a double for the metric being defined. That method will use the properties of the JavaClass and JavaModel set which are defined by the API of the model. The other method is a simpler one, and is intended to let metrics define their name, as it should appear in the View column.
This abstract class then has a method which returns a string representation of the metric value, rounded to an appropriate number of decimal places, and this method can be used by the View tab to display the metric values in tabular format.

There exists a `MetricsList` class which has in it a static list of defined metrics. The View tab takes each metric in this list, initializes it with information about which class to run that metric on, and the project context it is running in (i.e. set of `JavaModel`s), and then invokes the `compute` method to find and display the metric value. This is repeated for every class and every metric.

### 9.2 Defined Metrics

The metrics which are included in the CoderAider implementation are as follows. WMC, DIT, NOC and CBO are defined originally by Chidamber and Kemerer [5].

- **Number of attributes** - self explanatory.
- **Number of methods** - self explanatory.
- **Weighted Methods Per Class (WMC)** - Sum the individual complexity values of each method (in our case - the number of statements in the method body). The ratio of this sum and the number of methods indicates the average complexity and therefore possibility of reuse and maintainability. This metric may be used to indicate where methods have become too complicated, and could indicate the necessity for an “Extract Method” refactoring (see Listings 2.1 and 2.2).
9.3 Extensibility

- **Depth of Inheritance Tree (DIT)** - The depth of inheritance of a class, or the maximum length from superclass to subclass given multiple inheritance. The theory is that the deeper the class is in the hierarchy, the more methods it will inherit, making it more complex to manage.

- **Number of Children (NOC)** - The number of immediate subclasses of a class. The basis for this comes from the fact that a larger number of subclasses indicates greater possibility of reuse, i.e. a better design.

- **Coupling Between Object Classes (CBO)** - The count of the number of other classes to which a class is coupled. This is calculated by looking at all classes that are referenced by a class. Excessive coupling can be hazardous to design and prevents reusability. Therefore this figure should be minimized in most cases (although some design patterns encourage coupling for various reasons).

- **LCOM2** - LCOM2 is defined as:

\[
1 - \frac{\sum \psi}{\mu \cdot \alpha}
\]  

where \( \mu \) is the number of methods in a class, \( \psi \) is the number of methods that access a particular attribute, \( \sum \psi \) is the sum of \( \psi \) over all attributes of a class, and \( \alpha \) is the number of attributes in a class.

LCOM2 is the percentage of methods that do not access a specific attribute, averaged over all attributes in the class. If the number of methods or attributes is zero, LCOM2 is undefined and displayed as zero.

- **LCOM3** - LCOM3 is defined as:

\[
\frac{\mu - \sum \psi}{\mu - 1}
\]  

where \( \mu \) is the number of methods in a class, \( \psi \) is the number of methods that access a particular attribute, \( \sum \psi \) is the sum of \( \psi \) over all attributes of a class, and \( \alpha \) is the number of attributes in a class.

LCOM3 varies between 0 and 2. Values 1-2 are considered alarming. In a normal class whose methods access the class's own variables, LCOM3 varies between 0 (high cohesion) and 1 (no cohesion). When LCOM3 = 0, each method accesses all variables. This indicates the highest possible cohesion. An LCOM3 value of 1 indicates an extreme lack of cohesion. In this case, the class is a good candidate to be split. When there are variables that are not accessed by any of the class's methods, \( 1 < \text{LCOM3} \leq 2 \). This happens if the variables are dead or they are only accessed outside the class. Both cases represent a design flaw. The class is a candidate for rewriting as a module. If there are no more than one method in a class, LCOM3 is undefined. If there are no variables in a class, LCOM3 is undefined. An undefined LCOM3 is displayed as zero.

**9.3 Extensibility**

This framework now makes it easy for other developers to write their own metric and plug it into CoderAider, and take advantage of the model generated by CoderAider. The developer of a new metric should extend the `AbstractMetric` class, define the `getValue` and `getName` methods, use the model API to find out relevant information about the class being analysed, and then add the new metric object to the `MetricsList`. Unfortunately, the major limitation of this approach is that the plugin must be recompiled before the changes will take effect in a workspace. Removing the need to recompile the plugin is left as a possible extension to this project, as it falls outside the scope. This limitation is not a severe one, and a possible way to resolve it may be to create an XML style metric definition which is loaded dynamically at runtime.
This chapter has shown the design of a metrics framework which may be used for two purposes:

- To improve the accuracy of the algorithm which matches the example code in the refactoring specifications, to the user's codebase.
- To enable the user to see various metrics for their own classes, and possibly extend CoderAider to display their own metrics too.

From the GUI perspective, clicking on either a package or class refreshes the metrics view, as shown in Figure 9.2. This is a tab at the bottom of the screen. If a class is selected, it will only have one row, otherwise if a package is selected, it will have one row per class of that package. Each class has defined for it a number of metrics. The numerical value of each metric (rounded appropriately) is shown in each column. The meaning of each of these numbers was discussed earlier in this Chapter.

We have now finished discussing the design and implementation aspects of CoderAider. The next chapter shall demonstrate some of the things which CoderAider can achieve.
Chapter 10

Testing

“Trying to improve software quality by increasing the amount of testing is like trying to lose weight by weighing yourself more often.”

- Steve McConnell

This chapter shall focus on the project development methodologies. We shall begin by focussing on functionality testing, and scalability testing. We will demonstrate the capabilities of CoderAider by using examples, some of which stem from the refactoring literature discussed in Chapter 2, and some of which are unique to CoderAider, as other tools cannot perform such refactorings. A real world scenario will be shown where CoderAider was run on the source code of Apache Tomcat.

The difficulty when testing a system such as this is ensuring coverage. Various different scenarios need to be tested, and it is difficult to cover each and every possible scenario that may arise. The best that we can do is ensure all paths in the code are covered by any tests performed.

10.1 Strategy

The strategy used for developing this project was a form of test driven development. Before commencing the development of this project, I created a number of pieces of code which had obvious deficiencies. These were the test cases for CoderAider. CoderAider was regularly run on these test cases, and the results were examined manually. Two things were considered when examining the results:

- Whether or not the new code compiled correctly
- Whether or not the changes made to the new code were semantically valid (i.e. did they achieve their intended goals)

This second condition for semantic validity made it difficult to use automated testing tools to check whether tests were being satisfied or not.

Initially, development was done with the sole purpose of making as many test cases succeed as possible. This was to ensure that a running product was created in the minimal amount of time. After this phase, extensive refactoring was carried out on the project code in order to improve its design and efficiency. New test cases were added as and when required, especially when new functionality was being added. Whenever a new piece of functionality was added to CoderAider, or an existing piece of functionality was refactored, all tests were re-run, as a form of regression testing. This proved to be a highly effective method of finding exactly which changes during development were responsible for any observed irregularities. If any major discrepancies were found with the newest version of the code, I used my version control system (SVN) to rollback any changes, and include them one by one, in order to observe which change was responsible for causing tests to fail.
Benefits

Using this methodology had the advantage that I could develop code in small steps. As the developed code was being made to pass a test case, I could also be certain that all written code was being covered by a test case. Due to regular testing during development, defects could be eliminated quite early in the process, meaning that lengthy debugging sessions were not required.

Limitations

Using this form of test driven development means that the code can only be as good as the test cases written. In order to prevent this from being a limitation, I had to ensure that the test cases covered a variety of aspects, and were such that, if satisfied, would meet the aims of the project. Another limitation was that it was difficult to test the user interface parts of the project using test cases. This had to be done manually, at regular intervals.

10.2 Test cases

This section looks at the functionality offered by CoderAider, by demonstrating each test case. Note that throughout this section, when we say that CoderAider “used” the refactoring specifications outlined in the sections in Appendix A, we actually mean that CoderAider used the matching algorithm in Chapter 7 to find which refactoring specifications were applicable, and which regions of user code had potential problems, and then used the transformation algorithm in Chapter 8 to suggest an improvement based on the appropriately found refactoring specification(s).

The test cases described in this section were not chosen at random. I chose them carefully in order to ensure that both simple and complicated refactorings were being tested. The key things that need to be tested are:

- Adding new classes or deleting existing classes from the user’s codebase
- Adding new attributes or deleting attributes from a class in the user’s codebase
- Adding new methods or deleting methods from a class in the user’s codebase
- Adding new parameters to methods or deleting parameters from methods in the user’s codebase
- Changing the visibility and other modifiers of both attributes and methods in the user’s codebase
- Changing the return types of methods in the user’s codebase
- Changing the object types of the parameters of methods in the user’s codebase
- Changing the bodies of methods in the user’s codebase
- Changing / adding / removing superclasses and superinterfaces

Now we shall describe a few of the test cases used, to demonstrate the effectiveness of CoderAider.

10.2.1 Overridden Attributes

It is bad practice to override an attribute declaration. This is because the attribute is inherited anyway, so there is no need to override its declaration. Listing 10.1 shows the test code that was created. Listing 10.2 shows how CoderAider improved that code. Note that the overridden attribute has been deleted. Section A.1 shows the refactoring specification that was used by CoderAider to determine this course of action. Note that this specification is generic, and it will allow CoderAider to detect and fix overridden attributes of any object type, not just the String demonstrated in this example. To ensure this was the case, the same example was repeated with int as the attribute type. The results were identical, and omitted here for brevity.
10.2. Test cases

Listing 10.1: Overridden attributes test case

class OverriddenField {
    String s;
}
class Another extends OverriddenField {
    String s;
}

Listing 10.2: Improvement for overridden attributes test case

class OverriddenField {
    String s;
}
class Another extends OverriddenField {
}

10.2.2 Overridden Method calls in constructors

Often, it is a bad practice to call a method in a constructor, if the class can be extended. This is because the extended class may invoke the super constructor, but override the method called by the super constructor. This overridden method may use a field that has not been initialized yet, leading to an error, meaning that the extended class can never be constructed without error. This code example is shown in Listing 10.3. Listing 10.4 shows how CoderAider fixed this problem. In fact, it was unable to suggest a fix, and so inserted a comment highlighting the problem. This is the same approach suggested by the refactoring specifications. Section A.2 shows the templates used to specify this behaviour.

10.2.3 Improving a Singleton

In order to improve the design of a standard singleton pattern, we can introduce the initialization on demand holder idiom, as first suggested by Bill Pugh [31]. The implementation relies on the initialization phase of execution within the Java Virtual Machine (JVM). A private inner static class is introduced with a static variable which holds the instance of the singleton object. This variable is only initialized when it is needed, i.e. when a method refers to it. This is a form of lazy initialization, as it postpones the initialization phase until absolutely necessary. It also guarantees a thread safe singleton, as the JLS guarantees initialization to be serial, meaning that no further synchronization is required. This pattern is useful if the initialization of the class is expensive, or highly concurrent. The crux of the pattern is the safe removal of the synchronization overhead associated with accessing a singleton instance [46].

Listing 10.5 shows the test case that was created. Listing 10.6 shows how CoderAider improved it. Section A.3 shows the refactoring specification templates that were used to define this refactoring.

10.2.4 Removal of a null final field

Any attribute that is final and initialized to null should be deleted, as it clearly indicates a design fault. While this may cause non-compilation (as code may refer to this attribute), the compiler will merely be highlighting the areas of code that relied on an attribute that was null anyway. Listing 10.7 shows the test case that was created. Listing 10.8 shows how CoderAider improved it. Note the attribute has been removed, and the comments, other attributes and methods have remained intact. Section A.4 shows the
Listing 10.3: overridden method calls test case

class BadConstructor {
    public BadConstructor() {
        myMethod();
    }

    protected void myMethod() {
    }
}

class BadConstructorExtendor extends BadConstructor {
    private String itsAField;

    BadConstructorExtendor() {
        super();
    }

    public void myMethod() {
        itsAField.equals("name");
    }
}

Listing 10.4: Improved version of overridden method call test case

class BadConstructor {
    public BadConstructor() {
        // CoderAider: Constructors should not call overridden methods,
        // as this can prevent subclasses being instantiated.
        myMethod();
    }

    protected void myMethod() {
    }
}

class BadConstructorExtendor extends BadConstructor {
    private String itsAField;

    BadConstructorExtendor() {
        super();
    }

    public void myMethod() {
        itsAField.equals("name");
    }
}
Listing 10.5: Singleton test case

```java
public class BadSingleton {
    private static BadSingleton instance;

    public BadSingleton() {
    }

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

Listing 10.6: Improvement for Singleton

```java
public class BadSingleton {
    private static class SingletonHolder {
        private static final BadSingleton instance = new BadSingleton();
    }

    private BadSingleton() {
    }

    public static BadSingleton getInstance() {
        return SingletonHolder.instance;
    }
}
```
Listing 10.7: Null final test case

```java
public class NullFinal {
    public final String something = null;
    private int k = 2;

    /*
     * A random comment
     */
    public NullFinal() {
        k = 3; // Another comment here
    }
}
```

Listing 10.8: Improvement for null final test case

```java
public class NullFinal {
    private int k = 2;

    /*
     * A random comment
     */
    public NullFinal() {
        k = 3; // Another comment here
    }
}
```

refactoring specification templates that were used to define this refactoring.

10.2.5 Type Safety

The State pattern in Java is often used to model situations where the object can only be in one state out of a possible set of states. In these cases, the states are usually modelled as `String` attributes which are `static` and `final`. There is also a state variable which holds the current state of the object. This pattern is fine, but can present some difficulties if there are methods which directly modify the state of the objects. For example, a method may take a `String` as input and set the current state to be equal to this input, under the assumption that the input will be one of the possible set of states for the object. This assumption may not always hold, and rogue clients may cause the object to enter an invalid state.

Listing 10.9 shows the test case that was created. Note there are actually two separate problems with this code. Firstly, there is a null final field (as described in Section 10.2.4). Secondly, there is the type safety problem described here, as the `setState` method sets the variable which holds the state of the object (`thestate`) to whatever is passed into the method. Note also that this code is a slight variation of what may be expected - one of the state attributes is set to `private`, and the other is set to `public`. This shows the versatility of the recognition engine of CoderAider in being able to deal with such differences.

Listing 10.10 shows how CoderAider improves this code. There are a number of things to note here.

- The field which is `final` and is initialized to `null` has been deleted.
- A new class has been created to store the states.
- The fields `astate` and `bstate` previously used as the states have been removed, and re-used in the newly created class for the creation of state objects.
Listing 10.9: Type safety test case

class AClass {
    String thestate;

    private final static String astate = "astate";
    public final static String bstate = "bstate";
    public final String a = null;
    private int k = 3;

    public AClass() {
        k = k + 1;
        int j = k - 2;
        thestate = bstate;
        k = k + 2;
    }

    public String getState() {
        return thestate;
    }

    public void switchMethod() {
        if (thestate.equals(bstate)) {
            thestate = bstate;
        }
    }

    void setState(String a) {
        thestate = a;
    }
}

• The type of the attribute which stores the current state of the object (thestate) has been changed to match the newly created class which will be holding the states.

• The constructor included a statement which modified the current state of the object by assigning to it. This has been updated to use the setState method, and the reference to the state has been correctly updated to use the new representation of states (AClassGood2.bstate). Note that the position of this statement has remained unaltered within the constructor.

• The getState method used to return the current state of the object. This was previously a String. However, as the type of the current state has changed to AClassGood2, the return type of this method has been changed as well.

• Most importantly, the object type of the setState method’s parameter has now been changed to AClassGood2, i.e. the newly created object for representing states. This ensures that type safety is achieved, and that rogue clients cannot cause the object to enter into an invalid state.

Section A.5 shows the refactoring specification templates that were used to define this refactoring. Note that for this particular example, as there were two kinds of flaw, CoderAider will have applied the refactoring specification templates in Section A.4 as well.
Listing 10.10: Improvement for Type safety test case

class AClass {
    AClassGood2 thestate;
    private int k = 3;

    public AClass() {
        k = k + 1;
        int j = k - 2;
        setState(AClassGood2.bstate);
        k = k + 2;
    }

    public AClassGood2 getState() {
        return thestate;
    }

    public void switchMethod() {
        if (getState().equals(AClassGood2.bstate)) {
            setState(AClassGood2.bstate);
        }
    }

    public void setState(AClassGood2 a) {
        thestate = a;
    }
}

class AClassGood2 {
    private String name;

    public static final AClassGood2 astate = new AClassGood2("astate");
    public static final AClassGood2 bstate = new AClassGood2("bstate");

    public String toString() {
        return name;
    }

    private AClassGood2(String r) {
        this.name = r;
    }
}
10.2. Test cases

Listing 10.11: Sample source code which is using the chain of responsibility pattern

```java
class MyClient {
    String MethodName () {
        MyHandler k = new MyHandler();
        return "k";
    }
}

class MyHandler {
    MyHandler successor;

    int handReq(Object sro) {
        successor.handReq(sro);
        return 0;
    }
}

class MyConcreteHandler extends MyHandler {

    int handReq(Object sro) {
        super.handReq(sro);
        return 3;
    }
}

class MyConcreteHandler2 extends MyHandler {

    public int handReq(Object sr) {
        super.handReq(sr);
        return 2;
    }
}
```

10.2.6 Chain of Responsibility

For completeness, we shall demonstrate the use of CoderAider to find areas of source code where design patterns have been used. These areas can be flagged for review - this may be beneficial for learning purposes. Alternatively, this may be useful to illustrate to the developer whether their code is conforming to design patterns. If the developer was not sure about the design used, this may provide a useful pointer for further research about how to go about implementing the pattern. The pattern we shall demonstrate is the chain of responsibility pattern. Listing 10.11 shows the user’s source code. Listing 10.12 shows how CoderAider updated it. This shows that CoderAider was able to recognise the various elements of the user’s source code which were similar to the participants in the chain of responsibility pattern defined in the refactoring specification. Section A.6 shows the design pattern template (refactoring specification) that was used to specify this functionality.
Listing 10.12: CoderAider analysed code for the chain of responsibility pattern

class MyClient {
    String MethodName () {
        MyHandler k = new MyHandler();
        return "k";
    }
}

class MyHandler {
    MyHandler successor;
    int handReq(Object sro) {
        //CoderAider: Template_Handler class, handleRequest method
        successor.handReq(sro);
        return 0;
    }
}

class MyConcreteHandler extends MyHandler {
    int handReq(Object sro) {
        //CoderAider: Template_ConcreteHandler class, handleRequest method
        super.handReq(sro);
        return 3;
    }
}

class MyConcreteHandler2 extends MyHandler {
    public int handReq(Object sr) {
        //CoderAider: Template_ConcreteHandler class, handleRequest method
        super.handReq(sr);
        return 2;
    }
}
10.3 Demonstration on Apache Tomcat

Apache Tomcat is a servlet container used in the official reference implementation of Java servlets, and Java Server Pages \(^1\). It is one of the most popular platforms for deploying Java based web applications. In such applications, both security and stability is of great importance.

The Tomcat project was selected in order to test the effectiveness of CoderAider at finding and correcting design faults, or suggesting methods for improvement. The source code for Tomcat is readily available. It has also been made available as a preconfigured Eclipse project, as part of the CoderAider distribution, for demonstration purposes. See Appendix B for details on obtaining this set of sources.

10.3.1 Area of analysis

Apache Tomcat consists of 112 packages and 1296 classes. CoderAider was used to analyse a few of these packages. To demonstrate a typical bug found and fixed by CoderAider, we shall consider the org.apache.tomcat.util.http package. Within this package, CoderAider was able to uncover an interesting discovery. There exists a class here called BaseRequest. According to the comments, the function of this class is a “general purpose object for representing HTTP requests”.

10.3.2 Problem

The problem unearthed by CoderAider was the Type Safety problem, as discussed in Section 10.2.5. See Listing 10.13 for the original source - only the relevant parts have been shown. The BaseRequest class has an attribute declared as String scheme. It also has two public final static attributes, namely SCHEME_HTTP and SCHEMEHTTPS. The only assignments to the attribute scheme are one of these two static attributes, meaning that scheme is the attribute which holds the current state of this object, and is expected to only take one of these two values. Interestingly, there is a setScheme method which takes a String as a parameter, and assigns it to the current state, i.e. the scheme attribute. This is prone to the same problems discussed earlier, namely that a rogue client can cause this object to enter an invalid state by calling the setScheme method with an invalid state.

It seems that clients can use this class to create a request which has any protocol, and not just the default HTTP and secure HTTPS. The developers of this class in Tomcat have overlooked this scenario, and therefore the implementation may be prone to attack if clients set the protocol to be something other than the two intended ones (HTTP or HTTPS).

Note that we cannot be certain whether or not this is intended behaviour, but given the evidence, there is a high probability that this problem has been overlooked.

10.3.3 Solution

Using the refactoring specifications outlined in Section A.5, CoderAider was able to suggest a way to fix the problem found above. The changes performed were as follows.

- It deleted the two public final static attributes, namely SCHEME_HTTP and SCHEMEHTTPS.
- It changed the type of the attribute which holds the current state (scheme) to the type of the new class which is to be created (BaseRequestGood2).
- The class has a method called recycle which assigns a value of SCHEME_HTTP to the attribute scheme. As the object type of the attribute scheme has changed, and the attribute SCHEME_HTTP no longer exists, CoderAider has changed this to be scheme = BaseRequestGood2.SCHEME_HTTP instead.
- It changed the return type of the getScheme method, as it was returning the scheme attribute, which has already had its object type changed.

\(^1\)Apache Tomcat - http://tomcat.apache.org
Listing 10.13: Original flawed Tomcat source code

```java
public class BaseRequest {
    // scheme constants
    public static final String SCHEME_HTTP = "http";
    public static final String SCHEMEHTTPS = "https";
    ...
    String scheme;
    ...
    public void recycle() {
        ...
        scheme = SCHEME_HTTP;
        ...
    }
    ...
    public String getScheme() {
        return scheme;
    }

    public void setScheme(String s) {
        scheme = s;
    }
    ...
}
```

- It changed the method `setScheme`'s parameters from `String` to `BaseRequestGood2`. This will ensure type safety, as clients can only pass an object of type `BaseRequestGood2` into the method, and so the attribute `scheme` can only have a valid state assigned to it.

- It created a new class called `BaseRequestGood2`, which has a `private` constructor, meaning that only the states `SCHEME_HTTP` and `SCHEMEHTTPS` can be created, meaning that rogue clients cannot construct a new invalid state (protocol).

See Listing 10.14 for the fixed code - only the relevant parts have been shown.

### 10.4 Summary

This chapter has demonstrated both the flexibility and scalability of CoderAider in finding and fixing problems in the user's code. It has shown several examples of refactorings which existing tools cannot perform in an automated fashion. In addition, we have also seen how CoderAider is able to cope with code which varies from the refactoring specifications. This chapter has therefore highlighted some benefits of the approach adopted by CoderAider towards refactoring specification and execution.

This concludes the explanation and demonstration of CoderAider. In the next chapter, we shall step back and assess to what extent the aims of this project have been met, and how limitations have become apparent during development.
Listing 10.14: Automatically fixed Tomcat source code

```java
public class BaseRequest {
    ...
    BaseRequestGood2 scheme;
    ...
    public void recycle() {
        ...
        scheme = BaseRequestGood2.SCHEME_HTTP;
        ...
    }
    ...
    public BaseRequestGood2 getScheme() {
        return scheme;
    }
    ...
    public void setScheme(BaseRequestGood2 s) {
        scheme = s;
    }
    ...
}

class BaseRequestGood2 {
    public static final BaseRequestGood2 SCHEME_HTTP = new BaseRequestGood2("SCHEME_HTTP");
    public static final BaseRequestGood2 SCHEME_HTTPS = new BaseRequestGood2("SCHEME_HTTPS");
    private String name;
    
    public String toString() {
        return name;
    }
    
    private BaseRequestGood2(String r) {
        this.name = r;
    }
}
```
Chapter 11

Evaluation

“The first 90% of the code accounts for the first 10% of the development time. The remaining 10% of the code accounts for the other 90% of the development time”

- Tom Cargill

We shall now look at some of the contributions of this project towards the refactoring community, and assess the effectiveness of the techniques implemented. This project has developed a unique methodology for the representation of refactorings which is both flexible and extensible. We have seen that it can be used to easily specify refactorings which current tools are unable to perform. This project has also contributed a simple model which can be used for numerous purposes such as analysing code, transforming code, generating metrics, analysing examples provided by refactoring designers, and so on. We should now look at how effective these elements are towards meeting the aims of the project.

11.1 Evaluation Criteria

The evaluation criteria for this project are largely similar to the aims discussed in Chapter 3. In this section, we shall evaluate the extent to which each aim has been satisfied.

11.1.1 Interaction with existing development practices

This aim has been met fairly well. This is because:

- The plugin created works in Eclipse which is a popular IDE for development in Java.
- Plugins for Eclipse are easy to install and use on existing Java projects, meaning minimal effort on the part of the developer.
- The plugin created does not need any further configuration after installation in order to be used.

However, there is a limitation of this project. That is that it can only work as an Eclipse plugin, on a project already configured in Eclipse. Those developers that do not use Eclipse, and prefer other IDEs for Java such as Netbeans will find it a nuisance to use CoderAider, as they would firstly have to setup their project in Eclipse (so that it compiles properly), and then install the CoderAider plugin. This is a definite area for improvement. Perhaps CoderAider could be a standalone application which can be invoked by thin plugins in either Eclipse or Netbeans.

11.1.2 Minimal human intervention

This was stated as an aim intended at increasing the usability of the product. As required, CoderAider does not require any of the developer’s source code to be annotated or for it to conform to any particular
standard in order for analysis to be carried out. In this way, this aim has been met. This enables for the rapid application of CoderAider to existing development projects without any administrative or monotonic code labelling or formatting.

However, with the wisdom of hindsight, this may seem like a limitation. Scenarios occur where CoderAider would greatly be helped if the developer chose to provide further details about his intentions. For example, CoderAider wouldn’t have to guess which patterns the developer intended to use if the developer provided that information in input - perhaps as code annotations. Lack of this feature is not a great limitation, but only a minor possible improvement, as CoderAider is still capable of functioning effectively without it.

11.1.3 Detection & Correction

Existing automated tools often rely on the user’s input to determine the areas where a refactoring should be applied, or to determine which refactorings should be applied, as discussed in Section 2.7. CoderAider attempts to fulfill all three stages of the refactoring process, namely:

1. Refactoring opportunity identification - by using the refactoring specifications which specify a generic version of a “bad” coding pattern (or an antipattern). The procedure developed was explained in Chapter 7.

2. Refactoring selection - by using the refactoring specifications which specify the corrected “good” version of the “bad” code, the transformations to be applied can be determined. The procedure developed was explained in Chapter 8.

3. Refactoring application - by applying the sequence of transformations determined in the previous step, also explained in Chapter 8.

The testing carried out in Section 10.2 shows the effectiveness of the detection and correction of coding defects. As discussed, a number of tests were carried out which used code which wasn’t quite the same as the code examples in the refactoring specification. This seemed like an appropriate test methodology as real world source code will never match the refactoring specifications identically. As shown in that chapter, CoderAider was able to correctly apply the appropriate refactoring specification to variants of the code, as well as to exact matches of the code. This means that the aim of refactoring opportunity detection has been met fairly well. In addition, the tests demonstrate that once a specification (or specifications) have been chosen for use on the source code, CoderAider was able to correctly calculate the sequence of transformations required, and apply those transformations onto the source code to produce improved source code (as illustrated by the numerous corrected examples listed in Section 10.2). At a high level, this satisfies the main aim of this project, which was to automate the detection and correction of code defects. A few other points worth evaluating:

- Behaviour preservation or improvement - this is difficult to evaluate. CoderAider merely followed the instructions given the refactoring specification. It did not attempt to evaluate the specification itself for correctness, as it was assumed that the refactoring designer knew the implications of the specification. This might be disadvantageous in the long run, as more and more users contribute and share specifications with each other. Designers may write specifications intended for use on a single project only, and other designers may use those specifications on their projects too without realising the full extent of their implications. In this case, some form of behavioural analysis would have been useful. However, this aspect of CoderAider is not necessarily a disadvantage. Sharing refactoring specifications in a community is the intended future for the evolution of CoderAider. It is capable of incorporating new specifications into the analysis process with ease, without the need for recompilation. Encouraging such sharing can only improve the effectiveness of CoderAider.

- Syntactic correctness is achieved by CoderAider, yet it is not explicitly checked for. There is a precondition that the refactoring specifications will not introduce syntactic errors into the code.
• Compilation errors are avoided as far as is possible by CoderAider. It takes care to ensure that transformations made will not cause compilation faults, by extrapolating from the intended transformations any additional transformations that would be needed in order to prevent compilation failures. This is handled by the transformations wrapper outlined in Section 8.2. However, successful compilation is not guaranteed by CoderAider. As CoderAider works on a level much higher than other refactoring tools (e.g. on the method, class and package level), the extent of code which is affected by its changes is also greater. If CoderAider is only invoked on a single class, the changes it makes there may be beneficial to that class, but may also need additional changes to other classes as well. This is a limitation of CoderAider, as it will not analyse the other classes to find out the potentially affected classes as a result of this transformation (as the user has only invoked it on a single class, and told it not to analyse the other classes). However, given the test cases developed, this limitation of CoderAider did not seem to pose any immediate performance problems, even on larger projects such as Tomcat, where it was able to correctly determine and fix a problem, and ensure that there were no compilation errors.

11.1.4 Operational Aspects

The aims outlined in this section were mainly intended to improve the usability of CoderAider, as well as develop a level of trust with the user. As such, these aims were broadly achieved:

• If required, users can selectively disable certain refactoring specifications from being applied during the analysis. This choice is easy to make, and the user is not obliged to change the default selection in order to start analysis.

• CoderAider presents a comprehensive changelist of intended changes, as shown in Figure 4.11. Here, the user can select which classes to apply the changes to, and can view a detailed visual diff style view of the changes being made. This satisfies the aim of making changes clear to the user.

• As shown by Figure 4.6, CoderAider integrates itself with the history in Eclipse, so that any changes made can be easily reversed. Once again, the aim of easy reversal is satisfied. Testing revealed this feature to work as expected, and quirks such as incorrect reversal were not observed during the testing process.

Overall, the operational goals of CoderAider have been met fairly well. As discussed earlier, the major limitation is the lack of support for IDEs other than Eclipse. Operationally, this may be beneficial in order to encourage a greater level of support for CoderAider. A front end for managing refactoring specifications has not been included in CoderAider. It was felt that this was an unnecessary requirement, as the refactoring specifications are expected to be annotated classes which compile properly, meaning that developers would probably prefer to use Eclipse for creation of refactoring specifications.

11.1.5 Flexibility

This aim intended that the specification method for refactorings be generic, and that variations to the specification should also be dealt with appropriately. The levels to which this has been satisfied has already been discussed briefly.

The specification methodology developed enables developers to write examples of “bad” and its corresponding “good” code, using annotations to increase the expressiveness of each specification. This has proven to be a fairly effective approach, as the tests performed in the previous chapter have shown. A wide variety of different types of refactorings can be performed by CoderAider. Things like removal of overridden fields, and highlighting of the dangers of calls to overridden methods in the constructor are fairly simple examples of the flexibility of CoderAider. A more complex example is the detection of type safety. Section 10.2.5 shows the amount of work which CoderAider is able to perform on this topic, given the fairly simple refactoring specification which shows the “bad” and “good” versions of the problem. No other refactoring tool has the power to carry out both the detection and correction of problems such as type safety. This is a major achievement of CoderAider. Not only can CoderAider execute this detection
and correction with ease, it is also flexible enough to recognise and correct variations of this problem and
detect when this problem has been mixed with another problem (such as the unnecessary final field
initialized to null, as shown in Section 10.2.5). What further adds to the flexibility of CoderAider is its
extensibility, discussed in the next section.

Despite the ability of CoderAider to enable the specification of complex refactorings, which other tools
are not capable of, and identify and execute them appropriately, the approach adopted by CoderAider
does lack some flexibility. The most important assumption made by CoderAider is that refactorings that
improve code on the design level, or class level, can always be specified by use of examples of “bad”
code, and its corresponding “good” code. While this is often the case, it is unfortunately not always the
case. It is conceivable that examples might arise which cannot be represented in simplistic terms such
as “bad” and “good” code. CoderAider tries to account for this fact by the use of templates as critics
(see Section 5.3.4), and this does mitigate the weakness to the extent that the code defect can be made
visible even if it cannot be corrected using this form of refactoring specification. However, this approach
still fails to cater for the case when the code defect cannot be expressed in terms of a generic “bad” code,
even with the use of the annotations available. See Figure 11.1 for a simple example of what cannot
be expressed in CoderAider using the current methodology. This highlights a need for other techniques
to be used in conjunction with CoderAider, namely heuristics such as metrics to detect opportunities
to perform an “Extract Method”, and a dataflow component to correctly perform this refactoring. We
have already designed an extensible metrics framework, and integrating this further into the CoderAider
refactoring lifecycle would have beneficial results.

11.1.6 Extensibility

The major goal of CoderAider was to enable the detection and correction of coding defects. Extensibility
was added by use of a novel concept for refactoring specification - writing examples of “bad” and “good”
code in Java, and using annotations to add some extra information. Developers can build up a library of
such examples, share them with each other, and apply them on their own projects. New examples can be
applied to an existing CoderAider installation by dropping them into the specifications folder - there is
no need to recompile CoderAider, or even to restart the Eclipse workbench, as they are processed every
time the CoderAider analysis is invoked. The details of this approach towards refactoring specification
have been discussed in Chapter 5. The major advantages of this approach are that developers can share
domain specific code examples with each other - it will quite often be possible to create a refactoring
example suitable for one domain only, and sharing these with other developers in the same domain will
prove useful. Another advantage is that it is easy to define new code examples by extracting code from
existing projects to leverage existing knowledge.

Another aspect towards evaluating the extensibility of this plugin is considering its components. A
major component is the code model. As discussed in Chapter 6, this model uses the AST provided by
the Eclipse JDT and creates a semantic wrapper around it, which makes it easier to rationalise about
code as if it were actually code, rather than a tree with nodes and leaves. This model also encapsulates
the modification functionality provided by the Eclipse JDT, so that clients may use the model to perform
modifications easily, without having to use to the complex JDT framework. The model is composed of a
generic aspect, and a Java-specific aspect, so that if required, support for other languages can be built
into CoderAider as well. To show the effectiveness of this model component, Chapter 9 discusses a new
framework built around this model which can be used to determine various code metrics. It seems that
the model created is good enough to support at least two different uses - namely metric calculation,
and code analysis for problem detection and correction. The metrics framework created, to demonstrate
the reusability of the model, is itself extensible as previously discussed. Developers should find it quite
straightforward to add their own metrics to that framework. Unfortunately, this will need for CoderAider
to be recompiled, and the Eclipse workbench to be restarted, and this is one of the current limitations of
that framework.

Let us now turn to the limitations of CoderAider in terms of extensibility. As was previously discussed,
CoderAider assumes that refactorings can be specified using “bad” and “good” code, and as such, only
supports the definition of refactorings which conform to this assumption. This limits its extensibility, as

```
11.1. Evaluation Criteria

Figure 11.1: A simple refactoring which CoderAider cannot express or perform
it cannot perform other types of refactorings. The infrastructure of CoderAider is also rather dependent on this assumption, meaning that it will not be easy to modify the source code to support other kinds of refactorings. However, this does not include refactorings such as those carried out at the code level. While CoderAider does not include a dataflow analysis component, or perform any detailed analysis at the code level (i.e. the method bodies), the architecture is extensible enough to allow the insertion of such a stage at a later date.

11.1.7 Scalability

One of the aims of the project was to try and ensure that the final result works on not only test examples, but also real projects. The tests conducted earlier show that CoderAider has no problems running on test examples. A test was also conducted on Apache Tomcat (see Section 10.3). CoderAider was able to successfully determine a probable code defect, and suggest a fix for it. The severity of this defect was not great, but it could mean that clients of the class in question can cause it to enter into an invalid state (i.e. cause it to use an invalid network protocol). This shows that this aim has been satisfied as well, although it should be noted that it is quite difficult to assess the full success of this aim with a limited number of refactoring specifications. At the moment, there are only six specifications which have been included in CoderAider. It is hoped that over time, developers will contribute their own ones for use by CoderAider, but it does mean that at the moment, it is difficult to say how well CoderAider will perform with a larger number of specifications upon which to draw.

On a side note, a test was conducted which introduced the same code defect (type safety) twice into a project. CoderAider was able to detect both occurrences of this defect, and fix them appropriately, ensuring that the fixes did not conflict with each other. This shows that CoderAider does have scalability in terms of finding and correcting the same defect multiple times, and that it is able to correctly apply the refactoring specifications concurrently. However, the stability and efficiency of CoderAider still cannot be assessed properly without the creation of further refactoring specifications.

11.1.8 Improvement in code quality

It is difficult to judge whether CoderAider has contributed a remarkable improvement towards code quality. It works on the principle of leaving the transformations to the refactoring designer, meaning that it is only as effective as the refactoring specification designer. This limitation may be overcome if designers start sharing their refactoring specifications with each other. Indeed, if a library of specifications does become available in the future, then given the evidence observed during testing, it seems that CoderAider should be fairly effective at improving code quality, or at least providing pointers as to where quality may be improved.

It is difficult to use metrics to evaluate code quality. While a metric may provide a vague idea of the quality, it is by no means accurate, especially in cases where the size of the class(es) being refactored is very small. Therefore, we cannot use metrics to assess the improvements provided by CoderAider. In fact, if we did use metrics to assess the quality of the suggestions, we would merely be assessing the quality of the refactoring specifications currently in use, rather than the quality of the CoderAider framework itself. This does once again underline the underlying feature of this project, which is that the refactorings performed will only be as good as the refactorings specified prior to analysis. The extensibility of the plugin means that a great number of refactorings can be specified prior to analysis, and more can be added as and when needed, meaning that in theory, the refactorings performed should be of fairly good quality.

CoderAider analyses the given refactoring specifications in serial order. This has the advantage that all of the specifications provided will be applied fully, and that those that are run towards the end of the analysis phase will benefit from the improvements provided by those that were run towards the beginning of the analysis phase. However, the limitation of this approach is that the analysis does not take into account conflicting refactorings. In theory, it is possible to create two refactoring specifications which have completely opposite effects (one may delete an attribute, the other may add an attribute). The approach used by the analysis will mean that no changes are made to the code (as the changes made by
the first specification will be undone by the changes made by the second specification). This behaviour
seems to be a fair expression of what is acceptable. However, in the case where there is a greater conflict,
we may well end up with the situation where CoderAider decreases the quality of the code as a result
of poor refactoring specification design. This scenario should be rare, as it won’t be too often that two
designers have completely conflicting views about a design problem, and even if they do, they should
take care when publishing the refactoring specifications for CoderAider and inform the users about the
potential for conflict.

11.2 Comparison with existing tools

11.2.1 ArgoUML

ArgoUML (Section 2.4.1) is a UML modelling tool which has critics. These critics look for things like
design pattern faults, and other rather simplistic heuristics such as classes with too many methods. It is
possible to extend this framework and specify your own critics, but this is a lengthy process, and involves
a lot of changes to existing code. Some peculiar coding conventions have to be followed too.

The extensibility of CoderAider is better than that of ArgoUML. Adding new refactorings involves
writing examples of code, and saving the files to a default folder. The plugin does not have to be
recompiled, and the new refactorings are picked up immediately. This is much better than the rather
convoluted methods imposed by ArgoUML of modifying code, recompiling, and ensuring that any newly
written critics follow certain coding conventions.

ArgoUML can only deal with the singleton pattern at the moment, and it seems difficult to add
support for new design patterns. Adding such support in CoderAider involves writing out an example
usage of the pattern only. As such, it already supports the singleton and chain of responsibility patterns,
and it is easy to add more. The details of how a critic should analyse code are sketchy in ArgoUML. In
CoderAider, it is simple - just write an example of some “bad” code, using annotations if you wish, and
write its corresponding “good” equivalent.

Therefore, overall, it is fair to conclude that the functionality offered by CoderAider is easier to use,
and has greater variety, than that of ArgoUML. However, there are a few things that ArgoUML can do,
but CoderAider cannot - namely syntax and style analysis. CoderAider was never intended to look at
the low level code and its style, and this is a point for improvement.

11.2.2 PMD

PMD (Section 2.4.2) is a static code analysis tool for Java, and is capable of finding several hundreds of
code level defects, mainly limited to those defects that occur in the code inside method bodies rather than
the design defects of classes. PMD is easier to extend than ArgoUML, and one can either write Java code
or an XPath expression to define a new refactoring. However, both methods involve knowledge of the AST
used to represent code. This makes it rather complicated to write your own refactorings. CoderAider
presents an alternative method of extensibility which removes this need to know about the AST and
how code is represented internally. It also removes the need to reason about code transformations
programmatically, as all transformations needed will be found by CoderAider. While this does make
CoderAider easier to use and extend, it also limits the expressiveness slightly. PMD can express much
more in terms of lower level code defects such as inefficient logical conditions for if statements, but
CoderAider only has minimal support for such bug fixes, as it concentrates on higher level entities such
as classes, attributes and methods. Ideally, the low level capabilities of PMD would be combined with
the higher level capabilities of CoderAider to produce a more complete solution.

11.2.3 FindBugs

FindBugs (Section 2.4.3) is a similar tool to PMD, but works on the bytecode level. To write bug
detectors, some understanding of the structure of JVM bytecode and of class files is required. The BCEL
and FindBugs libraries handle some of this task, extracting information from the bytecode and presenting
it at a slightly higher level. Unfortunately, the documentation on both the BCEL and FindBugs support for dismantling a class is lacking, meaning that the main disadvantage of FindBugs is that it is difficult to create new bug definitions. As noted earlier, this is one of CoderAider’s strongpoints, as the definition and installation of new refactoring specifications is very easy.

FindBugs has a dataflow component, which means that it can find a greater variety of bugs than PMD. CoderAider does not have a dataflow component. A dataflow component would help with the analysis of code, but this was outside the scope of CoderAider, which aimed to look at the higher level aspects of code. This is a limitation, but as previously discussed, it should be possible to extend the infrastructure of CoderAider to include a code level analysis component without too much hassle.

11.2.4 Jackpot

Jackpot (Section 2.4.4) is quite similar to CoderAider. Both use the concept of templates to specify refactorings, which makes the specification and extensibility of the system easier. Jackpot uses the notion of metavariables ($\text{name}$) to denote real variables or names in the source code being analysed. CoderAider is similar, but uses real Java code instead of metavariables. Both approaches use code similarity techniques to find matches, and transform them to be better, using predefined specifications of what is “better”. This means that both approaches have similar benefits and limitations, and these have already been discussed previously.

What differentiates CoderAider from Jackpot is the same thing that differentiates CoderAider from a lot of other existing refactoring tools - ability to specify and perform refactorings on the class as a whole (class, attributes and methods) rather than on fragments of code.

11.3 Summary

Let us now recap some of the benefits and limitations of CoderAider. Some observed benefits are:

1. Ability to specify new refactorings with ease, using a novel approach involving giving examples of “bad” and “good” code.
2. Ability to extend the system with minimal effort.
3. Ability to specify domain specific refactorings, and share them with other developers easily.
4. Ability to extract code from existing projects, and use it directly to specify a new refactoring.
5. Ability to perform refactorings on a much higher level than other tools - for example at the class, method and attribute level rather than at the code level.
6. Ability to perform a wide variety of high level refactorings - for example, the type safety problems observed in Apache Tomcat is very difficult to detect using other tools.
7. Ease of integration with Eclipse.
8. Flexibility in specifying refactorings, to ensure that variations to the “bad” code examples provided are also picked up on, and actioned properly.

Some limitations of CoderAider are:

1. Lack of a dataflow component.
2. Lack of analysis of code at the low level code (such as analysing constructs and checking the logic of if statements).
3. Lack of integration with other IDEs such as Netbeans.
4. False positives may still be generated.
Chapter 12

Conclusions & Future Work

“A computer is a stupid machine with the ability to do incredibly smart things, while computer programmers are smart people with the ability to do incredibly stupid things. They are, in short, a perfect match.”

- Bill Bryson

This project was motivated by the fact that software maintenance has relatively few tools which can be considered as being very useful. Refactoring was studied as a means of making code easier to understand, or improve its design, making it a useful addition to the software maintenance process. This project has developed a novel methodology which makes both the specification and application of refactorings much easier, especially at on higher level entities such as classes. This process works by using real code examples to demonstrate good or bad practices, or desired design patterns - an innovative and exciting approach not currently offered by any other tool. This project also tried to reduce the level of human interaction needed in the refactoring process by automating the refactoring opportunity identification, refactoring selection and refactoring execution stages. Such integration, using the techniques outlined in this report, has paved the way for a whole new line of research, and there is scope for plenty of further work, as described later.

I originally intended to make available a set of refactoring specifications which were very general and could be potentially applied to any project. During the course of development, it became clear that it was very difficult to create any generic refactoring specifications applicable to all projects. In order to overcome this challenge, I ensured that the core principle upon which the project was built was extensibility - this enables developers working in certain areas (such as driver development, or web services) to create their own refactoring specifications and share them with others in the same domain. The unique methodology I have proposed in this project also makes it simple to write these refactoring specifications, or extract them from existing code, in order to better leverage the knowledge of existing projects. This approach has offered advantages such as intuitiveness and powerful expressiveness, and underlined the importance of extensibility to any such system.

While developing the methodology, flexibility was possibly the most important challenge, as I had to ensure that refactoring specifications made up of example code would not suffer the fate of being overly specific and applicable to very few situations. I had to ensure that the algorithms developed were capable of using the example code in the refactoring specifications, and being able to apply the concepts of that example to other projects, even if they did not match the example entirely. I believe that the approach I adopted of using code annotations in the examples to extend their scope worked reasonably well - in fact it scaled well too, and it was even able to find and correct a design fault in Apache Tomcat, which is a large (1296 classes) project. I think that in the future, this approach needs to be extended further to incorporate more annotations in order to better the matching process. If it weren’t for time constraints, this project should have also looked at supplementing the methodology contributed with other traditional means such as data flow analysis, duplicate code detection, and heuristical analysis, in order to improve the detection of code defects.
In short, CoderAider is able to perform a variety of structural code fixes which existing tools cannot, and therefore promises to be a useful addition to a developers toolkit. Its capabilities can only increase further once its user base starts increasing and people start sharing code examples with each other and extracting code examples from their existing projects, especially from projects tied down to certain design principles because of their domain of interest. Most importantly, CoderAider has the flexibility to expand its capabilities further in this manner with ease.

12.1 Future Work

There are numerous enhancements which could be carried out in the short and long term to this project. These can broadly be categorised into two areas - refinements and new directions.

12.1.1 Refinements

These extensions are mainly refinements to the project, or areas which could do with some further consideration.

1. Creating a larger database of refactoring specifications would help to make the project usable on real projects, as it would become capable of finding and fixing a larger number of problems.

2. Adding a dataflow analysis component to find problems such as dead code or to perform things like constant folding.

3. Adding a code-level analysis component to find problems at the lower code level (such as inappropriate use of libraries, unsafe casting etc).

4. Extracting the functionality of this project out into a standalone application, which can be invoked from an Eclipse plugin, or a Netbeans plugin.

5. Integrate the tool with a graphical UML view to show how changes are being made structurally - useful for learning and documenting purposes.

6. Adding support for more languages apart from Java - such as C++ and C#. This should be feasible, as the model used does have a generic component which can be reused. However, the bulk of the logic that deals with matching code patterns, and performing transformations on them, is specific to Java, because it needs to be so closely involved with the code. This logic may need to be rewritten to support other languages.

7. Adding a tool to support the development of refactoring specifications. This is not really a priority, as developers can use Eclipse to create the refactoring specifications (as they are just Java code examples, with annotations, and which should compile properly using the standard Java compiler). If a tool was developed to aid this process, it may provide hints or tips while developing, or it might analyse what has been created to ensure it doesn’t conflict with existing specifications.

8. Features such as duplicate code detection will be useful, and fairly straightforward to add using standard algorithms.

9. Another potential area for improvement is the way that refactoring specifications are defined. More annotations can be developed to further increase the expressibility of the specifications.

10. Further integration of metrics with the code analysis techniques already implemented will be beneficial to decrease the likelihood of false positives.
12.1.2 New Directions

These extensions are much broader in scope, and indicate possible future areas that could be the subject of more research.

1. Performing behavioural analysis on the code to see how its behaviour is being modified, and attempting to ensure that side effects will be kept to a minimal.

2. Analysing the performance of the code to see if it may be reorganised to improve performance - for example, it may be possible to achieve the same goal of the code using fewer objects, or fewer loop iterations. Perhaps some form of loop fusion techniques could be beneficial, or even re-ordering the sequence in which a loop is performed in order to improve performance.

3. Extending support for J2EE, servlets and java server pages. While these can all use the rules for normal java design, there are specialised rules we can apply to these in order to improve the quality of their design.

4. Analysing the design from a security viewpoint. For example, native methods in Java have a lot of potential to cause damage such as corrupt memory, or security leaks, as they are a way of avoiding the Java API and so the security features of the JVM. It should be possible to analyse these scenarios to find regions which may be potentially harmful.

12.2 Final Thoughts

We have seen how the CoderAider methodology is potentially of benefit in the highly active research area of software maintenance, and in particular, refactoring for code improvement. It is quite difficult to make any judgement about how successful this approach is likely to be in general, but the initial results have proven to be very promising, and there seems to be a great deal of potential to help developers working in different domains. Further research in this area, particularly by combining the CoderAider methodology with more traditional methodologies will be very interesting indeed.
Appendix A

Templates

“Try to learn something about everything and everything about something.”

- Thomas H. Huxley

This part of the appendix outlines the various refactoring specification templates that are included by default in the installation of CoderAider. Users and refactoring designers can use the examples and style of these templates to create similar refactorings.

A.1 Overridden fields

A.1.1 Template for bad code
See Listing A.1.

A.1.2 Template for good code
See Listing A.2.

A.2 Overridden method calls within constructors

A.2.1 Template for bad code
See Listing A.3.

A.2.2 Template for good code
See Listing A.4.

A.3 Improving the Singleton

A.3.1 Template for bad code
See Listing A.5.

A.3.2 Template for good code
See Listing A.6.
Listing A.1: Template for bad code for overridden fields

```java
@ClassAnnotation( extend = Extends.SPECIFIED )

class Bad1 {
    @FieldAnnotation(
        visibility = {
            Visibility.DEFAULT,
            Visibility.PROTECTED,
            Visibility.PUBLIC
        },
        retType = ReturnType.ANY,
        change = false
    )
    Object instance;
}

@ClassAnnotation( extend = Extends.SPECIFIED )
class Bad2 extends Bad1 {
    @FieldAnnotation(
        visibility = {
            Visibility.DEFAULT,
            Visibility.PROTECTED,
            Visibility.PUBLIC
        },
        retType = ReturnType.ANY
    )
    Object instance;
}
```

Listing A.2: Template for good code for overridden fields

```java

class Good1 {
    @FieldAnnotation(
        visibility = {
            Visibility.DEFAULT,
            Visibility.PROTECTED,
            Visibility.PUBLIC
        },
        retType = ReturnType.ANY,
        change = false
    )
    Object instance;
}

class Good2 extends Good1 {
}
```
Listing A.3: Template for bad code for overridden method calls

```java
@ClassAnnotation(change = false, extend = Extends.SPECIFIED)
class Bad1 {
    @MethodAnnotation(change = false)
    Bad1() {
        method();
    }
    @MethodAnnotation(
            visibility = {
                Visibility.DEFAULT,
                Visibility.PROTECTED,
                Visibility.PUBLIC
            },
            parameters = Parameters.ANYMORE,
            returnType = ReturnType.ANY,
            change = false
        )
    void method() {
    }
}

@ClassAnnotation(change = false, extend = Extends.SPECIFIED)
class Bad2 extends Bad1 {
    @FieldAnnotation(change = false)
    private String somevar;
    @MethodAnnotation(change = false)
    Bad2() {
        super();
    }
    @MethodAnnotation(
            visibility = {
                Visibility.DEFAULT,
                Visibility.PROTECTED,
                Visibility.PUBLIC,
                Visibility.PRIVATE
            },
            parameters = Parameters.ANYMORE,
            returnType = ReturnType.ANY,
            change = false,
            Uses = {"somevar"}
        )
    void method() {
    }
}
```
Listing A.4: Template for good code for overridden method calls

```java
@ClassAnnotation ( change = false )
class Good1 {
    @MethodAnnotation (  
        comment = "// CoderAider: Constructors should not call overridden methods, \n        as this can prevent subclasses being instantiated.",  
        change = false  
    )
    Good1 () {
        method ();
    }

    @MethodAnnotation (  
        change = false  
    )
    void method () {
    }
}

@ClassAnnotation ( change = false )
class Good2 extends Good1 {
    @FieldAnnotation ( change = false )
    private String somevar:

    @MethodAnnotation (  
        change = false  
    )
    Good2 () {
        super();
    }

    @MethodAnnotation (  
        change = false  
    )
    void method () {
        somevar.equals("something");
    }
}
```
A.3. Improving the Singleton

---

Listing A.5: Template for bad code for Singleton

```java
@ClassAnnotation()
class Bad1 {
    private static Bad1 instance;

    private Bad1() {
    }
    @MethodAnnotation(
        visibility = {
            Visibility.DEFAULT,
            Visibility.PROTECTED,
            Visibility.PUBLIC
        }
    )
    public static Bad1 getInstance() {
        if (instance == null) {
            instance = new Bad1();
        }
        return instance;
    }
}
```

---

Listing A.6: Template for good code for Singleton

```java
@ClassAnnotation()
class Good1 {
    private Good1() {
    }
    private static class SingletonHolder {
        private static final Good1 instance = new Good1();
    }
    public static Good1 getInstance() {
        return SingletonHolder.instance;
    }
}
```
A.4 Removing final attributes initialized to null

A.4.1 Template for bad code

See Listing A.7.

A.4.2 Template for good code

See Listing A.8.

A.5 Type Safety

A.5.1 Template for bad code

See Listing A.9.

A.5.2 Template for good code

See Listing A.10.

A.6 Chain of Responsibility

See Listing A.11.
Listing A.9: Template for bad code for type safety

```java
@ClassAnnotation()
class Bad1 {
    @FieldAnnotation(quantity = Quantity.MANY)
    public final static String R = "R";

    String state;

    public Bad1() {
        state = R;
    }

    @MethodAnnotation(
        visibility = {
            Visibility.DEFAULT,
            Visibility.PROTECTED,
            Visibility.PUBLIC
        },
        quantity = Quantity.MANY
    )
    public void anotherSetMethod() {
        state = R;
    }

    public String getState() {
        return state;
    }

    @MethodAnnotation(
        visibility = {
            Visibility.DEFAULT,
            Visibility.PROTECTED,
            Visibility.PUBLIC
        },
        quantity = Quantity.MANY
    )
    public void method() {
        if (state.equals(R)) {
            state = R;
        }
    }

    @MethodAnnotation(
        returnType = ReturnType.ANY,
        visibility = {
            Visibility.DEFAULT,
            Visibility.PROTECTED,
            Visibility.PUBLIC
        }
    )
    void setState(String a) {
        state = a;
    }
}
```
Listing A.10: Template for good code for type safety

```java
@ClassAnnotation()
class Good1 {
    Good2 state;

    public Good1() {
        setState(Good2.R);
    }

    public void anotherSetMethod() {
        state = Good2.R;
    }

    public Good2 getState() {
        return state;
    }

    @MethodAnnotation(
        returnType = ReturnType.ANY,
        quantity = Quantity.MANY
    )
    public void method() {
        if (getState().equals(Good2.R)) {
            setState(Good2.R);
        }
    }

    @MethodAnnotation(
        returnType = ReturnType.ANY,
        quantity = Quantity.MANY
    )
    public void setState(Good2 a) {
        state = a;
    }
}

@ClassAnnotation()
class Good2 {
    @FieldAnnotation(
        quantity = Quantity.MANY
    )
    public final static Good2 R = new Good2("R");

    private String name;

    private Good2(String r) {
        this.name = r;
    }

    public String toString() {
        return name;
    }
}
```
Listing A.11: Template for chain of responsibility

```java
@ClassAnnotation ( extend = Extends.SPECIFIED )
class Template_Client {
    @MethodAnnotation {
        Uses={"Template_Handler"},
        returnType = ReturnType.ANY
    }
    void MethodName () {
    }
}

@ClassAnnotation ( extend = Extends.SPECIFIED )
class Template_Handler {
    Template_Handler successor;
    @MethodAnnotation {
        returnType = returnType.ANY,
        comment = "// CoderAider: Template_Handler class, handleRequest method"
    }
    void handleRequest (Object sro) {
        successor.handleRequest (sro);
    }
}

@ClassAnnotation ( quantity = Quantity.MANY, extend = Extends.SPECIFIED )
class Template_ConcreteHandler extends Template_Handler {
    @MethodAnnotation {
        returnType = returnType.ANY,
        comment = "// CoderAider: Template_ConcreteHandler class, handleRequest method"
    }
    void handleRequest (Object sro) {
        super.handleRequest (sro);
    }
}
```
Appendix B

Downloads

Here we list the items of CoderAider that are available for download from http://www.doc.ic.ac.uk/~nd103/CoderAider.

- Report.pdf - this is an electronic version of this report.
- CoderAider jar file - You can drop this into your eclipse plugins directory, and it will be run automatically the next time Eclipse is run.
- CoderAider src - This is the Java source code for CoderAider.
- Apache Tomcat src - This is a preconfigured project in Eclipse for the Apache Tomcat source code. This project can be opened in Eclipse, and CoderAider can be run on the org.apache.tomcat.util.http.BaseRequest class, as demonstrated in Section 10.3.
- Test cases - This is an Eclipse project containing all of the demonstration examples shown in this report. CoderAider may be run on all of them to see how it is detecting and fixing problems.
Bibliography


