A Model Driven Approach for Refactoring Heterogeneous Software Artefacts

Keith Dooley

A dissertation submitted in partial fulfilment
of the requirements for the degree of
Master of Science
at
Maynooth University.

Faculty of Science & Engineering, Department of Computer Science

February 2016

Head of Department: Dr. Adam C. Winstanley

Supervisors: Dr. Rosemary Monahan, Dr. James F. Power

Contents

List of Tables

List of Figures

Abstract

Refactoring is the process of transforming a software system to improve its overall structure while preserving its observable behaviour. Refactoring engines are normally used to perform these transformations for efficiency and in order to avoid introducing behavioural changes into the program due to human error. Although these engines do not verify that behaviour is preserved, it is widely accepted that automated transformations are less likely to introduce errors in comparison to manual refactoring. Despite the advantages provided by refactoring engines they fall foul of certain weaknesses.

Here we hypothesise that Model Driven Engineering can be used to produce improved refactoring engines that are less vulnerable to those weaknesses. We develop a Domain Specific Transformation Language for defining new composite refactorings from a set of built–in primitives and to script their application. We also develop an interpreter for the language, effectively providing an operational semantics, in the guise of an extensible transformation framework. We evaluate our approach with a case study examining the correlation between actual and predicted measurements of the Coupling Between Objects metric for classes that undergo the extract class refactoring. The results show that our approach is promising.

Chapter 1

Introduction

This chapter presents the motivation for the research presented in this dissertation. It states explicitly our thesis and provides an overview of our contributions. It concludes with a 'roadmap' that discusses the organisation of the remaining chapters.

1.1 Motivation

Opdyke (1992) presented refactoring as a methodical approach to improve the *structure* of a software system's source code while preserving its observable behaviour. It is often carried out by developers as an intermediary process, so that adding new functionality to the system is less laborious, or during preventative maintenance, so that the codebase remains comprehensible. Refactoring is an inevitable process for software that often changes, such as what Lehman and Fernández-Ramil (2006) categorise as 'E' type software. The process of refactoring involves applying structural transformations to the code and consists of two stages. The first stage is checking that a set of conditions hold and the second stage is actually performing the transformation. The purpose of the conditions is to ensure that the refactoring preserves the program's behaviour. The set of conditions applicable to a refactoring varies for each transformation and for the programming language that is used to implement the system under refactoring. We call atomic transformations *primitive refactorings* and when composed these form *composite refactorings*.[1]

Although manual refactoring is possible, and done often, it is generally accepted that it is more susceptible to undesired side effects due to human error. Many of these cannot be detected by the compiler so, although the system compiles, its behaviour will have changed. This leads to unpredictable behaviour that is costly to debug and rectify. If the errors are not detected until the system is released to the client then there may be other, potentially irreversible, consequences such as data loss.

Refactoring engines support the refactoring process and offer a semi–automated approach. Interaction between the developer and the refactoring engine is minimal. The developer guides the engine regarding *what* refactoring to apply and *where* it is to be applied. There is no longer a need for the developer to check that the necessary conditions are satisfied or that all the relevant areas of the codebase are updated. Despite the benefits afforded by refactoring engines they fall foul of these shortcomings, which we refer to throughout this dissertation as the four Is[2]:

[1] It will be clear from the context when we use the word refactoring to mean the activity of refactoring or a specific transformation.

[2] These are numbered so later when we refer to I1, for example, we mean the first in this list.

1. Inaccurate — The development of a complete and sound set (i.e. all conditions are considered and all conditions are correct) of refactoring preconditions is a non–trivial task and is dependent on the programming language being refactored. This claim is supported by Brett et al. (2007) who have shown that even the most widely used refactoring engines are susceptible to having overly weak or strong preconditions. There is no support for altering the preconditions once the refactoring engine has been compiled so users cannot fix erroneous conditions in 'the field'.

2. Inextensible — Refactoring engines offer limited refactorings to the user. Users cannot define their own refactorings and save them for later application. Popular refactoring engines offer only the primitive refactoring operations, which the user must compose time and time again to perform more useful refactorings.

3. Inflexible — Most refactoring engines are capable of refactoring just a single software artefact: source code. Nowadays this is no longer sufficient because software engineers work in an environment that consists of multiple heterogeneous artefacts. For example, there are artefacts that represent high level design documentation such as Unified Modelling Language (UML) class diagrams and artefacts that might even be involved during the system's runtime, such as database or Extensible Markup Language schemata. These are often interdependent and refactoring one requires changes to the others. Refactoring engines are limited in their capabilities in this respect. Some refactoring engines, such as that provided by the Eclipse IDE, are able to update some aspects of JavaDoc annotations while refactoring Java source code. However they do little more beyond this.

4. Improvident — Ge and Murphy-Hill (2014) suggest some developers ignore refactoring engines because they do not trust the changes refactoring engines make. We believe they continue to refactor manually because it enables them to see changes being made stepwise. This means that if the refactoring appears to be having a negative impact on other aspects of the system the developer can change his or her course of action. Common refactoring engines restrict the developer from doing this and changes are made irrespective of undesired effects. For example, refactoring engines will allow a developer to stupidly move all of a system's functionality into a single class despite the fact this degrades the quality of the system's design.

Our thesis, outlined next, addresses these shortcomings.

1.2 Thesis Statement

We believe there is a need to research and implement 'next generation' refactoring engines that are better suited for contemporary software engineering environments populated by heterogeneous artefacts. This sets the stage for our thesis:

> *Improvements to current refactoring engines can be made through an approach that ties together Model Driven Engineering (*MDE*) and formal methods (specifically Design by Contract (*DBC*) [3]) and encourages developers to define their own reusable*

[3]We note that the term 'design by contract' is trademarked but we use it in this dissertation when referring to approaches that share the same concepts.

and shareable libraries of refactorings.

We now briefly remark on the rationale for our thesis but delve into greater detail in Chapter 2.

MDE is an approach to software engineering where development is focussed on creating shareable models represented in some core format, which can be converted via model transformations into other representations. This might include source code. This is in contrast to earlier approaches where the goal is to represent classes of real world entities directly in source code. Interestingly, however, MDE has parallels with earlier Computer Aided Software Engineering (CASE) approaches. In the CASE approach developers modelled a system using a visual language such as UML and then forward engineered those models into source code. Changes made to the source code would then be reflected in the UML diagrams. The purpose was to keep the documentation and the code synchronised. The difference between MDE and CASE lies with CASE being only concerned with two representations. The general approach being taken remains the same. That is to say, there is a further step in the direction of developers working at higher levels of abstraction. MDE is discussed in greater detail in the following chapter.

DBC is a methodology for code development that has been successfully used to improve software reliability. Comparisons are commonly drawn between the approach taken in DBC and transactions in the business world. In the DBC approach, there exists a contract between the client who uses code in an arbitrary library and the provider of that library. The contract describes what conditions the client is obliged to fulfil in order for the code in the library to execute correctly. The correct execution of the code is also described in terms of conditions. The approach we propose to refactoring requires that developers define their own refactorings and the conditions that must be true about the *structure* of the program in order for the refactoring to be successfully applied. We briefly discuss how this will be done when discussing our contributions in the next section but a more in depth discussion is provided in the subsequent chapter.

1.3 Contributions

The work presented in this dissertation does not offer a complete solution to developing the next generation refactoring engines. However, we make the following contributions in order to evaluate our thesis. These are provided as prototype tools and a complete environment with the necessary set up and configuration is available by installing the VirtualBox disk image stored on the electronic medium disseminated with this dissertation.

- RefDstl Language — We have developed a prototype DSTL that can be used to specify and script refactorings on models of software systems in terms of a set of basic primitive operations. The language and requisite tools, such as a parser and text editor with syntax highlighting and content assist, are produced using MDE technologies. The notion of a DSTL also has roots in MDE. Design by contract features heavily in the language. The language targets the inaccuracy (I1) and inextensible (I2) shortcomings raised above. Although developing the necessary conditions is a difficult feat, we can at least give the end users the opportunity to weaken or strengthen the conditions being checked by the refactoring engine in 'the field' rather than having to look into the entire source code of the engine. Furthermore, refactorings written in the language can be saved to a script for later use.

- RefDstl Transformation Framework — This framework serves as the interpreter for RefDstl scripts. However, it is richer in capability than traditional interpreters. It is an extensible framework that allows others to plug in their own transformations (which are driven by refactorings) and allows artefacts other than source code to be refactored simultaneously. We have provided two noteworthy plug–ins for the framework among others. The first refactors the software system; it relies on the MoDisco[4] metamodel for representing Java systems. A model of the system under refactoring, conforming to the MoDisco metamodel, is transformed during the refactoring. When the framework is equipped with just this transformation then it behaves as a regular refactoring engine. The second plug–in enriches the framework with a notion of code quality. It allows developers refactoring to include requirements about the system's code quality in their refactoring conditions. This tackles the problem of improvidence (I4) described above. This plug–in uses the approach by McQuillan (2011). We perform a model transformation from the MoDisco model to McQuillan's metrics metamodel. This happens in real time while the system model is being refactored. We allow for plug–ins to be reasoned about in the refactoring conditions, which allows the developer who defined the refactoring to prevent refactorings being applied where the quality of the system (determined by metric measurements) deteriorates. By allowing further transformations to be plugged into the framework we permit flexibility. This addresses I3.

- Standard Library of Primitive Refactorings — We provide a standard library of primitive refactoring operations based on the same described by Opdyke (1992). It does not directly address the shortcomings addressed above but the need for it will become apparent later.

Figure 1.1 depicts a visualisation of how our framework operates. The portion developed by us includes what is inside the box entitled 'RefDstl Transformation Framework' and the RefDstl metamodel. We did not develop the MoDisco metamodel or contribute to the MoDisco project in any way. The diagram shows that the process of refactoring begins with a Java software system that is transformed into a model that conforms to the MoDisco metamodel. This can be done within an Eclipse Integrated Development Environment (IDE) equipped with the MoDisco plug–ins. The resulting model is then consumed by the RefDstl framework along with a script written in the RefDstl scripting language that conforms to the RefDstl language metamodel. The framework applies a sequence of transformations to the model of the system under refactoring. This yields one or more resulting output models, the most important of which is probably the refactored system that, again, conforms to the MoDisco metamodel. The refactored system can be transformed back into Java source code using a model–to–text transformation. However, this is outside the scope of this dissertation since we concern ourselves only with models that can be manipulated algorithmically.

We discuss the implementation and evaluation of these components throughout the dissertation. The 'roadmap' presented in the next section describes the structure of this document.

1.4 Organisation of Dissertation

The sequel is organised as follows:

[4]https://eclipse.org/MoDisco/

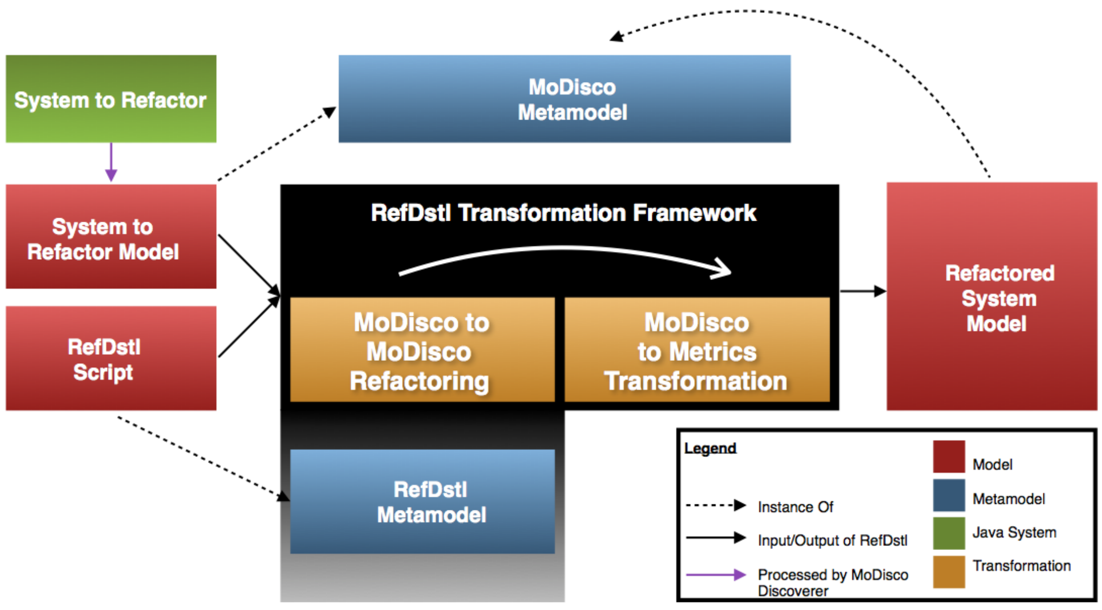

Figure 1.1: An overview demonstrating how the components of the refactoring system presented in this dissertation interact.

In Chapter 2, the necessary background material is provided. It addresses some of the assumptions that have been made thus far including the relevance of refactoring in software engineering circles. We also expand on our discussion of MDE and DBC. Furthermore, we discuss an alternative approach to ensure software quality, i.e. software metrics. That chapter comes to a conclusion with a discussion about how other researchers have used metrics to identify where refactoring should occur. This is followed in Chapter 3 with a discussion regarding the implementation of the RefDstl language. We discuss potential use cases for the language as well as language design goals. The implementation of the language is described and the chapter ends with language examples and a brief summary of the RefDstl standard library. In Chapter 4, we discuss the design and implementation of the RefDstl framework for interpreting RefDstl scripts as well as how to develop plug–ins that expand its capabilities. We evaluate the efficacy of our approach in Chapter 5 with a case study over a corpus of open–source Java software. Chapter 6 concludes the dissertation with a summary of our findings and a promise to investigate further avenues of research that arise from this body of work.

The appendices at the end contain supporting material. A list of acronyms is provided on page 71. In the electronic version of this document, the important terms and acronyms are hyperlinked to their expanded form. Appendix A contains the grammar of the RefDstl language to resolve any ambiguities concerning what is and is not a RefDstl program. Appendix B contains the full text of the RefDstl standard library.

Chapter 2

Towards a Model Driven Approach to Refactoring

In Chapter 1, we stated our thesis that MDE and DBC could be combined to address the shortcomings of current refactoring engines that we termed the 'four IS'. While our ultimate goal in this dissertation is to demonstrate the validity of this thesis, there are three research questions that we address prior to implementing a prototype sufficient for evaluation. We open this chapter by establishing those questions, which paves the way for the literature review that follows. This review allows us to identify weaknesses concerning the state–of–the–art and prevents us reinventing the wheel. The literature review covers the topics of: refactoring, DBC, MDE, and software metrics. This chapter also affords us the opportunity to introduce nomenclature and definitions.

2.1 Research Questions

The following questions drive the remainder of this chapter and the answers form the rationale for the implementation choices of our solution:

RQ1 What approaches have been taken to ensure that a system's behaviour is preserved post refactoring?

RQ2 What approaches have been taken to allow software engineers to define their own reusable refactorings?

RQ3 What considerations are paid to the overall quality of a software system while refactoring?

The connection between these questions and the 'four IS' should be apparent. RQ1 is related to I1, while RQ2 relates to I2. Both I3 and I4 are associated with RQ3.

We now examine the literature and begin with the topic of refactoring to address RQ1.

2.2 Refactoring

During its lifetime, a software system grows considerably to adopt and adapt to changing and new requirements. Evidence supporting this claim is offered by the data illustrated in Figure 2.1, which was collated from data gathered by Tempero et al. (2010). It shows that after passing twenty–three versions the Ant[1] software system grows twelve–fold from 20,832 to 255,690 lines of code. This is not an isolated event. All but one of the fifteen systems considered in the data gathered by

[1]http://ant.apache.org

Tempero et al. (2010) exhibits growth as it evolves. The ‘black sheep’ is JGraph[2], which reduces by an insignificant 176 lines of code.

Maintenance activities occur so frequently with software that it is estimated by Glass (2001) that they account for between 40% — 80% of software costs. In order to minimise the time, and consequently cost, of implementing changes in these rapidly growing systems it is highly important that the code is comprehensible to the responsible engineers. Approaches to software engineering can make writing comprehensible code quite a feat because many development approaches are iterative; from early approaches such as the spiral model by Boehm (1988) to more recent approaches such as extreme programming by Beck (1999). These iterative processes result in code being added repeatedly until the structure of the codebase begins to rot.

One solution to code rot is to engage in total productive maintenance (an idea admonished by Martin (2003)) — the code should be re–worked frequently to improve its structure. This however can result in regressions when defects are erroneously introduced. Refactoring is a methodical activity for improving code structure during which, code transformations are applied. However, these transformations are restricted to a set *(called refactorings)* that are widely believed to preserve observable behaviour.

Recent literature concerning refactoring has focussed on automating approaches to refactoring: the development of algorithms for identifying regions of the code that require refactoring, development and evaluation of refactoring engines that automatically apply refactorings and search based refactoring techniques that view the task of finding the best refactoring candidate as a search–space problem. Other work, closely related to RQ1, has focussed on proving the correctness of refactoring operations with respect to the observable behaviour of the program. These topics are surveyed in this section and an introduction to common refactoring operations is presented.

2.2.1 Refactoring Operations

Refactoring operations are the transformations applied to source code to improve its structure with the caveat that they preserve the observable behaviour of the program. Some of these operations can be viewed as atomic, which we call ‘primitive’ or ‘elementary’ refactorings while more complex and generally more useful molecular operations are known as ‘composite’ refactorings.

Primitive Operations & Behaviour Presentation

Primitive operations are the smallest atomic refactoring operations that can be applied to a program. The original body of work published on refactoring by Opdyke (1992) specified twenty–three primitives. These are listed in Figure 2.2. The intention of each should be self explanatory. However, the method for applying each of these is dependent on the programming language being refactored and how the program is represented. For example, if the program is represented as an abstract syntax tree then the refactoring will be performed using tree manipulations.

Not all primitive operations are relevant to every language. For example, Java omits pointers so the ‘convert instance variable to pointer’ refactoring is inapplicable. The operations were described by Opdyke (1992) with conditions that must be met regarding the structure of the program if the refactoring is to preserve behaviour. These are called preconditions. We draw attention

[2]http://www.jgraph.com

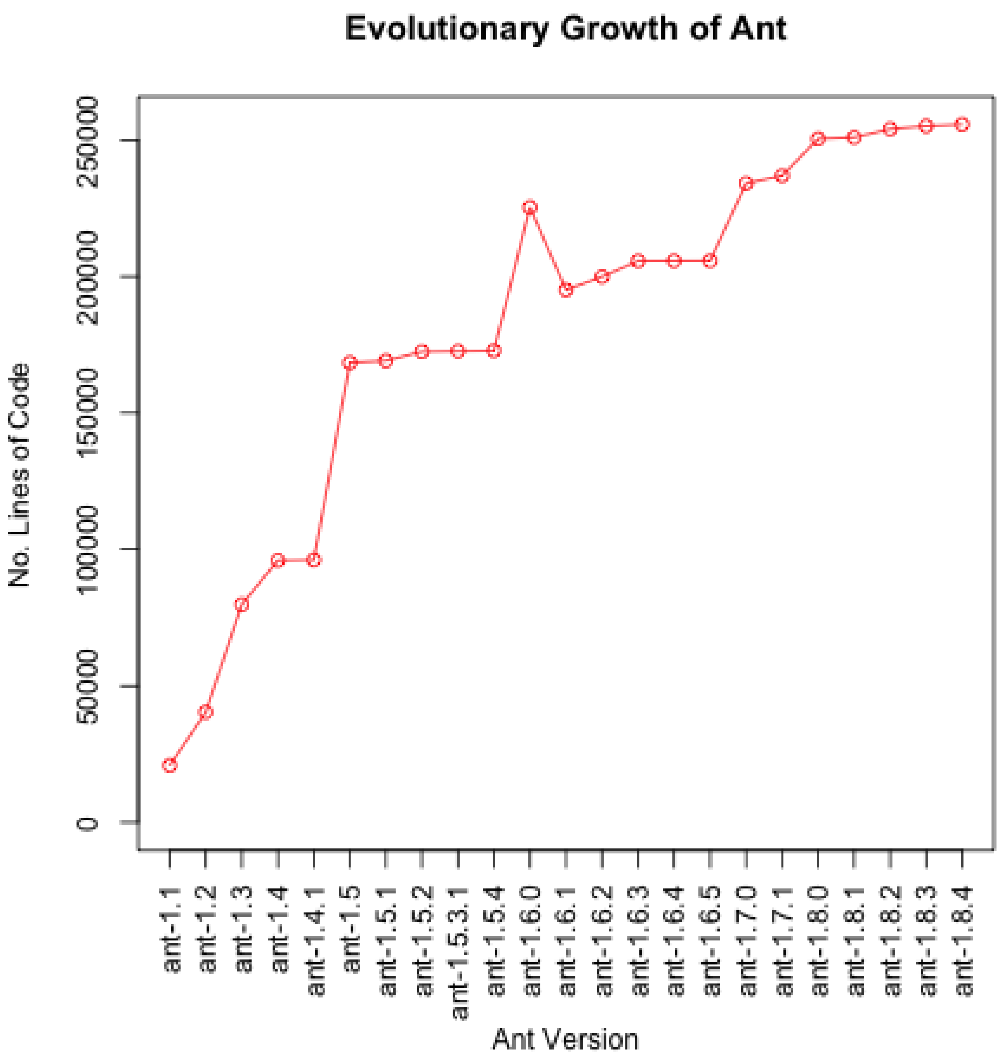

Figure 2.1: This graph shows how the size of the Ant software system grows twelve–fold in terms of lines–of–code between versions 1.1 and 1.8.4.

to the fact that the preconditions vary depending on the programming language being refactored, as do Mens et al. (2005). Furthermore, Mens et al. (2005) raise the point that it has never been demonstrated for mainstream programming languages that these preconditions are sufficient to guarantee that program behaviour is preserved for all refactorings. They claim that such a proof is impossible and we refrain from accepting this as a challenge here.

There have been attempts made with some degree of success to show that refactoring object–oriented *specification* languages (that have well defined semantics) does indeed preserve behaviour. For example, Carvalho Júnior et al. (2007) use CafeObj[3] to prove the correctness of extract, inline, and move method refactorings as well as self–encapsulate field in a sequential subset of Java named Refinement Object–Oriented Language (ROOL). The purpose of (ROOL) is to reason about 'object–oriented programs and specifications'. They encode the rules of the grammar

[3]http://www.cafeobj.org

of ROOL in CafeObj as well as the refactoring rules. CafeObj then 'mechanises' the refactoring proofs via rewriting.

Garrido and Meseguer (2006) take a similar approach using the Maude[4] algebraic rewriting system by Clavel et al. (2003). They provide an equational semantics in Maude for three Java refactorings and formal proofs for two: pull–up and push–down field.

Mens et al. (2005) use typed labeled graphs to represent programs and show how program refactorings can be represented as graph transformations where it is possible to prove that refactorings exhibit properties such as access, update and call preservation.

Soares et al. (2010) take a less formal approach to demonstrating that refactoring preserves behaviour. They developed the SafeRefactor tool. It performs a static analysis to identify methods that have common signatures in both the source and target programs. It then generates test programs to test these common methods using the Randoop[5] program and, if it finds any changes, it reports that a behavioural change has occurred.

Overbey et al. (2016) extend their earlier work (Overbey and Johnson, 2011) and use an approach to check behaviour preservation that they call 'differential precondition checking'. They concede that their approach is neither sound nor complete but they highlight that the core algorithm used in their approach is language independent and could be abstracted into a library for use in various refactoring engines. Unlike Soares et al. (2010), their approach allows for behaviour preservation to be tested before the refactoring is performed. Their approach involves representing the program under refactoring as a program graph (an idea borrowed from Mens et al. (2005)), which they explain as an abstract syntax tree with extra edges to represent semantic information, for example variable access. The refactoring is then simulated and an updated program graph is produced. The semantic edges of the two graphs are then compared for *expected* differences. We do not describe this further here but we believe that identifying what these differences should be is one of the weaknesses to their approach.

From the literature, it appears that proving refactorings preserve behaviour is an unsolved research problem. We believe that while this remains the case, refactoring engines should at least *move* their conditions away from the core code of the engine and allow these conditions to be altered by users 'in the field'.

Composite Operations

As the name suggests, composite refactorings are formed by composing primitive refactorings. For example, the extract class refactoring that features in the catalogue of refactorings by Fowler (1999) consists of a composition of create class, move field and move method primitives.

It is obvious that if each of the primitive refactorings are guaranteed to preserve program behaviour then composing any of these also preserves behaviour. However, Mens and Tourwe (2004) highlight work by Tokuda (2001) who describes certain compositions of refactorings as transactional. Transactional refactorings are those that when composed constitute a valid refactoring but if stopped at any point before completion then program behaviour would *not* be preserved and would therefore be invalid. They provide an example that they call 'delegate method across object boundary', which is the composition of a move method primitive and a create method accessor

[4]http://maude.cs.illinois.edu
[5]http://randoop.github.io/randoop/

- Create empty class.
- Create member variable.
- Create member function.
- Delete unreferenced class.
- Delete unreferenced variable.
- Delete member function.
- Change class name.
- Change variable name.
- Change member function name.
- Change type.
- Change access control mode.
- Add function argument.
- Delete function argument.
- Reorder function arguments.
- Add function body.
- Delete function body.
- Convert instance variable to pointer.
- Convert variable references to function calls.
- Replace statement list with function call.
- Inline function call.
- Change superclass.
- Move member variable to superclass.
- Move member variable to subclass.
- Abstract access to member variable.
- Convert code segment to function.
- Move class.

Figure 2.2: Primitive refactoring operations according to Opdyke (1992).

primitive (which was not listed by Opdyke (1992) and demonstrates that his list is incomplete). They provide a scenario that consists of a method a in class C, which is called by method b (also in class C). Method a is to be moved to class D. However, the move method primitive would not work in this situation because one of its preconditions requires that the method to be moved is not being referenced by any other methods. However, suppose we ignore this condition and carry out the refactoring along with the create method accessor refactoring. The program will be left in a state with method a located in class D and a new method a' in C, which is a delegate to the moved method a. This constitutes a valid refactoring when performed as a whole.

Choosing the correct refactoring to apply, whether it is a simple primitive or a composed sequence of primitives is a matter for the developer. Likewise, deciding when to refactor is also a decision to be made by the developer. However, there are some subjective notions to guide the developer along, such as code ‘smells’, which are discussed next. Software metrics also provide a more sensible approach. Their use is discussed in subsection 2.2.3 and more generally in section 2.5.

2.2.2 When to Refactor & Considerations for Developing a Refactoring Engine

Developers take two contrasting approaches for deciding when to refactor (Murphy-Hill and Black, 2008).

1. Developers who choose *root canal* refactoring restructure their source code after it has decayed.
2. On the other hand, developers who opt for *floss refactoring* improve their code structure in tandem with normal development activities.

Murphy-Hill et al. (2012) conclude that the latter is the favoured approach by developers.

Both approaches rely on indicators that suggest *when* and *what* to refactor. These indicators are referred to as ‘smells’ in the code. Fowler (1999) describes twenty–two such ‘smells’. These include ‘feature envy’ where the code in one class has a high number of dependencies on data in another class.

Tools known as ‘smell detectors’ have been developed to detect code ‘smells’, which can be used to determine when and what to refactor.

Simon et al. (2001) create a visual ‘smell detector’[6] to help developers identify move method, move attribute and inline/extract class opportunities. They extended the Crocodile tool (a language independent metrics tool) to show graphs where entities that are close according to a distance metric are presented close in Euclidean space. Their tool does not actually perform the refactoring but the refactoring should be performed manually on elements that are distant from each other.

Fokaefs and Tsantalis (2011) developed a plug–in for the Eclipse IDE named JDeodorant[7]. It detects the ‘God’ class code ‘smell’ and suggests remedying refactorings. The authors have used it to identify opportunities for applying the extract class composite refactoring, which is used as a solution for eliminating the ‘God’ class ‘smell’ (where one class contains too much data or behaviour). They evaluate the tool in an empirical study involving version 5.3 of JHotDraw. They evaluated their approach using a ‘professional software quality assessor’. Although they do not discuss what qualifies her/him with such a coveted title, s/he supported the suggested refactorings by the tool and said that nine out of the sixteen tools suggested that refactoring improved the code with respect to its comprehensibility.

Murphy-Hill and Black (2008) highlight that the problem with ‘smell detectors’, such as JDeodorant, is that they are unsuitable for developers who choose to floss refactor. They argue that because floss refactoring is an activity that is repeatedly done, tools that require explicit interaction are less useful. They suggest seven habits of good ‘smell detectors’. The first of these is availability: the tool should report ‘smells’ as early as possible and with as little involvement as possible from the developer. They also support unobtrusiveness: the tool should work without blocking the developer. They have implemented the tool ‘Ambient View’ that takes into consideration these and the remaining five suggested good habits.

We next discuss automated approaches to refactoring beyond ‘smell detection’.

2.2.3 Manual, Automated and Search–based Refactoring

Investigations by Murphy-Hill et al. (2012) suggest that refactoring tools are under utilised. From their study, they found that 89% of 145 refactorings (supported by tools) were performed manually. The developers in this study were ‘toolsmiths’ who produce the refactoring tools so they are unquestionably aware of the existence of appropriate aides. Manual refactoring can be carried out by carefully following the steps laid out in refactoring catalogues such as Fowler (1999). However, this approach is not recommended by us as manual refactoring has the potential to introduce defects due to human error. For example, suppose in some method there exists a local variable that shadows a global variable. If the developer renames that local variable in all but one place then the program will compile and execute but will use the incorrect variable at some point. We believe that refactoring should at the very least be done using refactoring tools such as the

[6] A misnomer perhaps but, nonetheless, an apt description.
[7] https://marketplace.eclipse.org/content/jdeodorant

refactoring tool provided in the Eclipse IDE. Although these tools should be more reliable than performing the refactoring steps manually, there are other automated approaches that can further increase efficiency, for example, search based refactoring.

O'Keeffe and Ó Cinnéide (2006, 2008a,b) have developed a search based approach to refactoring and implemented the CODEIMP automated design improvement tool. It is inspired by Harman and Clark (2004) who consider software metrics as fitness functions. It takes a Java 1.4 program and performs a search of the space of applicable refactorings and refactors the Abstract Syntax Tree according to the results. Moghadam and Ó Cinnéide (2011) later improved on this work by updating CODEIMP to be applicable to Java 6 source code.

We believe this full level of automation might be a step too far for many developers. We settle for a compromise that would allow for refactorings to be scripted. With this in mind, and also our suggestion that conditions should be extracted from the core refactoring code, we take a look at DBC. This relates to RQ2.

2.3 Formal Specification & Design by Contract

Safety critical systems require a level of rigour beyond the capabilities of traditional software testing techniques such as those described by Myers et al. (2011). For this reason, rigorous approaches to software development were introduced including Design by Contract.

In the formal specification approach to developing software, the system's critical components are modelled using specification languages such as the algebraic rewriting language Maude (Clavel et al., 2003). Properties of the system can then be reasoned about rigorously. For example, Listing 2.1 shows a specification written in Maude for a pelican–crossing[8]. The proofs at the end of the script demonstrate the safety of the system in the sense that it will not signal pedestrians to cross at the same time cars are instructed to proceed. It also demonstrates that the specification provides both cars and vehicles an opportunity to go. However, fairness is *not* reasoned about. Whilst Maude is described as an executable specification language, it is unlikely that it would be used to implement real world software. This is particularly true in this example, since pelican crossings would require an implementation language more suited for embedded systems, such as 'C'. This creates a chasm between the specification and the implementation.

To address this issue, Meyer (1997) introduced Design by Contract (DBC). It allows for parts of the program code to be annotated with statements that express the intent of the code. Static analysis tools reason about the code with respect to the specification and can determine (in many cases) if the implementation implies the specification. The specifications in DBC are commonly referred to as contracts because they describe the obligations of both the client using the code and of the developer who provides the code.

```
mod PELICAN-CROSSING is
   protecting BOOL .

   sort Globals .

   vars C P : Bool .
```

[8]This is a translation of the specification written in the Event–B language and published in the Rodin handbook (Jastram, 2014).

```

  op invariants : Bool Bool -> Globals [ctor] .

  crl [set_peds_go] : invariants(C, P) => invariants(C, true)
  if not C and not (P and C) .

  crl [set_peds_stop] : invariants(C, P) => invariants(C, false)
  if not (P and C) .

  crl [set_cars_go] : invariants(C, P) => invariants(true, P)
  if not P and not (P and C) .

  crl [set_cars_stop] : invariants(C, P) => invariants(false, P)
  if not (P and C) .
endm

*** If the following search returns no results then we are given some assurance
*** that the system is safe. It is not an absolute guarantee, the search might
*** have timed out before finding a result for example.
search in PELICAN-CROSSING
  : invariants(false, false) =>* invariants(true, true) .

*** If the following search returns non--empty results then we are guaranteed
*** that the specification allows for cars to have some opportunity to go.
search in PELICAN-CROSSING
  : invariants(false, false) =>* invariants(true, false) .

*** If the following search returns non--empty results then pedestrians are
*** guaranteed to be given an opportunity to go.
search in PELICAN-CROSSING
  : invariants(false, false) =>* invariants(false, true) .
```

Listing 2.1: A Maude specification for a pelican crossing system.

Contracts for specifying behaviour in object–oriented programming languages such as Java (using JML by Leavens et al. (1999)) involve formalising:

- Preconditions — These constrain the values that may be passed as arguments to the method. If the constraints are not held then the client cannot be guaranteed that the method will behave in the intended manner. Preconditions can be computed by working backward through the program using Hoare logic (Hoare, 1969).

- Postconditions — These make guarantees about the result of methods. Postconditions are only valid when the preconditions are met and are implied by the body of code in the method.

- Invariants — These are placed on classes or within loops. Invariants on loops exist only to facilitate the theorem prover in proving that the loop behaves correctly. Class invariants describe constraints about the state of the object at any time during its lifetime. Invariants are not considered in the remainder of this dissertation.

We believe that DBC goes some way to address RQ2. However, we still require an approach for representing programs being refactored. We believe that abstract syntax trees are too verbose for this purpose and we identify MDE as a solution. We are further motivated by MDE because it is also suitable for building a language to specify and script refactorings. We discuss MDE next.

2.4 Model Driven Engineering

The influence of modelling can be seen in many aspects of software engineering. At the high level, management use (descriptive) models called processes to describe how teams of engineers should structure their work activities. At a lower level, software engineers create models to 'represent something by something else' (Muller et al., 2012) such as classes to represent real world concepts. The benefits are apparent. Models allow concepts to be communicated clearly between developers or to serve as specifications for parts of a system to be implemented (we call these prescriptive models). Model Driven Engineering (MDE) builds on the advantages of using models by making the design and implementation of models the core activity of software engineering. MDE practitioners hold the view that 'everything is a model' (Bézivin, 2005). But what is a model? We accept the view expressed by Stachowiak (1973) (in German but translated in Muller et al. (2012)) that a model is *some thing* based on an *original* for a specific purpose that captures only the important properties that are relevant to that purpose.

It is often desirable to transform one model to another. This is done through a model–to–model transformation. For example, suppose we have a UML class diagram, the ability to transform this into a relational database schema, for example, or an XML schema is desirable. We might also want to transform the same UML class diagram into 'skeleton' Java code. This can be done with model–to–text transformations.

The success of modelling hinges on being able to describe models effectively. For this, the Meta Object Facility (MOF) is used. We briefly describe the architecture of MOF next.

2.4.1 Metamodel Hierarchy and Transformations

In order to describe a model we require an appropriate language. The four layer metamodel hierarchy exists to facilitate this. At the highest layer (M3) is the meta metamodel. It describes the kind of things that can exist in the M2 layer. We can think of the M2 layer as the actual language, for example, UML is located in this layer. In the M1 layer, we use the language of M2 to describe things, for example we might describe some class diagram. The M0 layer contains instances of what is described in M1. For example, it could be an object in the computer's memory that conforms to a class diagram described in M1. An object at M0 is *not* a Java class. A Java class also exists at M1, and is expressed by the Java language at M2 which is an instance of the grammar for Java at the M3 layer.

There is an interesting advantage to this hierarchy in that the M3 layer describes the minimal requirements for any language to be able to express the lower layers. For example, consider the example shown in Figure 2.3 (reproduced from (Nastov, 2013)). In this example, if the implementation language being used is powerful enough to express the concept of class at the M3 layer then the language is sufficiently powerful to describe the notions in the lower layers. This is true due to the 'instance of' relationship that crosses the boundaries of the layers.

However, having languages to describe models is pointless without concrete tools. MDE environments have been developed for this reason. We discuss these next.

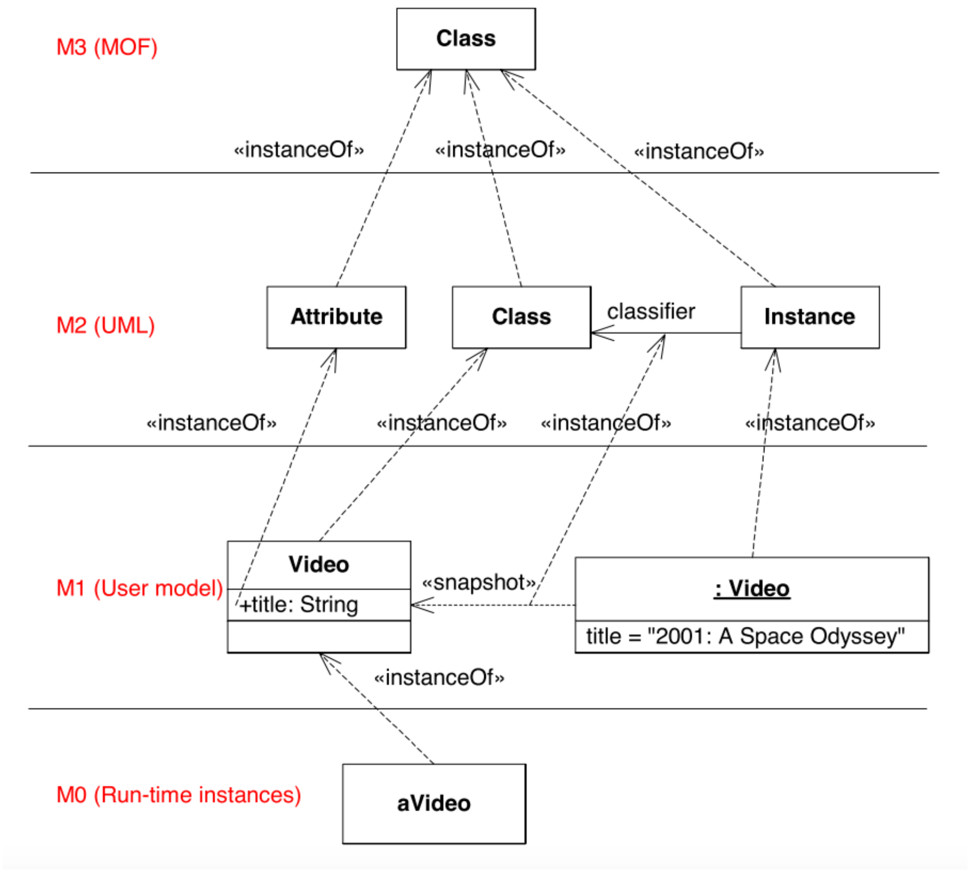

Figure 2.3: An example showing the four layers of the metamodel hierarchy.

2.4.2 MDE Environments

Frameworks have been developed specifically to accommodate MDE. The Eclipse Modelling Framework (EMF) (Steinberg et al., 2009) has been developed by the Eclipse foundation. It allows developers to build domain models either in Java, as XML schema definitions or as UML class diagrams. The models are mapped to a core modelling format called Ecore. This centralised model can be mapped to other model formats such as a generator model. The generator model can be used to produce skeleton Java code, code for testing and a plugin for Eclipse which provides an editor that has syntax highlighting and content assist.

Simulink by MathWorks[9] is an alternative environment for working with models. It differs from EMF with respect to its target audience. Simulink is suitable for engineers developing embedded systems. It provides a block diagram environment that allows code generation and simulation MathWorks. We do not mention more about it here as we are concerned with frameworks for *software* engineers.

MDE has been used to implement languages for modelling specific domains. We call these DSLs and discuss these next.

2.4.3 Domain Specific (Transformation) Languages

A domain specific language DSL is a language developed for a specific domain or task. The greatest difference between a DSL and a general purpose language is the vocabulary used. The keywords in a DSL will be words from the domain for which the DSL is designed. In a programming language, keywords will refer to programming constructs. Familiar examples of DSLs include the LaTeX language used for document preparation. Although it is Turing complete, its intended purpose is to describe the content and structure of articles, letters, presentation slides and other publishings. The TIKZ package for LaTeX (to produce diagrams such as Figure 5.2) is also a DSL so, hence, we can have one DSL embedded inside another.

DSLs are classified as being either:

1. Internal — An internal DSL is one that is developed within an already existing programming language. This is much the same as what traditionally have been called *libraries* but with a very focussed purpose. For example, Thompson (2011) presents a simple DSL for geometric manipulations of ASCII 'art' embedded in the Haskell[10] language. Internal DSLs are simpler to implement. For example, there is no need to implement a parser or interpreter for the language. It also has the benefit that users of DSL are not required to learn a new programming language syntax and can continue to use their preferred and familiar tool–chains (IDEs, text editors, etc.). We could also consider XML schema definitions, or XSDs, as crude internal DSLs since they describe sets of specific XML documents in XML.

2. External — External DSLs on the other hand are standalone languages, which come with their own advantages. Users of these languages do not need to have experience working with general programming languages. They only need an understanding of the domain in which they are working. This is beneficial because they are isolated from what could be obscure error messages while using general purpose languages. The implementation effort

[9]http://uk.mathworks.com/products/simulink/
[10]https://www.haskell.org

however increases. The language designer must develop their own parser and supporting tools such as an interpreter. However, language frameworks such as Xtext[11] and MPS[12] have been developed that relieve most of these burdens from the language designer.

Specific DSLs have been developed to transform models. For example, the Object Management Group (OMG) (OMG, 2015) specify:

1. Query/View/Transformation Operational — QVTO is an imperative language *specification* for transforming models. One implementation is ATL (Jouault et al., 2008). Although it has imperative features it also can be used declaratively, so, it also adheres to the QVTR specification.

2. Query/View/Transformation Relational — QVTR is a declarative language for performing model transformations. It relies on pattern matching to perform the transformations. The user describes a pattern that is matched on the input and this provides a rule for what the output should be.

Both QVTO and QVTR offer a general approach for describing model transformations. However, in some circumstances one might want to constrain the transformations that can be applied (as is the case with refactoring). We coin the term Domain Specific Transformation Language (DSTL) to refer to languages that exist to apply transformations under constraints.

2.4.4 Modelling Java Programs

The MoDisco[13] project has been created to help developers to modernise legacy software systems. MoDisco comes with a Java model in Ecore format for representing Java systems as well as knowledge discoverers that can be used to transform existing Eclipse Java projects into MoDisco models. It also comes with supporting tools such as a model browser for exploring the models visually in the Eclipse IDE and transformations to produce UML diagrams.

In this section, we have seen that models provide a practical approach to represent software source code and build languages (DSLs). It has also been employed in the area of software metrics, which relates to RQ3. We discuss this next.

2.5 Software Metrics

All branches of engineering rely on measurement to provide meaningful feedback. This can be used to assess quality or to make predictions (Meneely et al., 2012). Software engineering is no exception. For example, a software engineering team developing a system might test for the mean time between crashes. This can be used to evaluate the suitability of their product for specific scenarios.

At a lower level, the engineering team might evaluate the quality of their design. This can be used as an indicator to decide how easily the system can be maintained in the future. These measurements might even form part of the contract between the engineering team and the client.

[11]https://eclipse.org/Xtext/
[12]https://www.jetbrains.com/mps/
[13]https://eclipse.org/MoDisco/

Measurements could even be used as performance indicators that help management decide how to attribute specific portions of the system's implementation to specific engineers. For example, management might identify certain engineers as being better at implementing algorithms rather than object oriented design and vice versa. This trade off can be used to decide to whom tasks should be assigned.

Software metrics come in two varieties: project metrics and design metrics. We discuss both next before remarking on their validity and available suites developed for their use.

2.5.1 Project Metrics

Project metrics are high level metrics used to make management decisions. Examples include those published by Lorenz and Kidd (1994):

- Average person days per class — Suppose management is estimating the cost of staffing a new project and the high level design has already been completed so that management has an estimate of the number of classes that will be needed by the new system. Management can use data from previous projects to estimate the amount of time it will take to complete the system.

- Number of scenario scripts — A scenario script is a use case that documents actions that are performed, who they are performed by and who they affect. The number of use cases will indicate the size of the application and consequently the length of time required to implement it. Historical data is also required in this case.

Although we could devise project metrics to predict the time it would take developers to improve a system's structure, it is beyond the scope of this work and we do not address the issue further.

2.5.2 Design Metrics

Design metrics are used to evaluate the quality of an implementation. Three aspects of the system that are commonly measured include:

- Cohesion — Many software systems in existence are built using object–oriented languages that require a system to be split into modules called classes. A well designed object oriented system consists of classes that are semantically cohesive, i.e. each class represents just one meaningful concept. For example, a cohesive implementation of a 'person' class might have the person's name, age, address and phone number, but when loosely related details are added, such as bank account number and balance, then that class is no longer a cohesive unit. Many metrics related to cohesion consider the relationships between the methods and the attributes. A highly cohesive class should have methods that access a high number of the attributes. The exception to this rule of course are accessor and mutator methods. The literature contains a plethora of cohesion metrics that including the common LCOM metrics (Chidamber and Kemerer, 1991, 1994; Hitz and Montazeri, 1995) as well as the TCC and LCC metrics of Bieman and Kang (1995).

- Coupling — Coupling comes in two 'flavours'. The afferent coupling of one class is a measure of how dependent other classes are on it. High afferent coupling indicates that a class' interface is too public. On the other hand, and heavily related, is efferent coupling. This is a measure of how tied one class is to another. If a class *C* has high efferent coupling then it is considered to be poorly designed because it means that if the class or classes it relies on change then those changes must be reflected in *C*. Various coupling measurements have been proposed for this purpose including the CBO metric published by Chidamber and Kemerer (1991).

- Complexity — Most software engineers would agree that a program consisting of just sequential lines of code without iterations or conditions is less likely to contain defects than one with a high number of if–statements and loops. The rationale of course is that the developer might have forgotten to include a condition for some special case or written an incorrect guard on a loop. Programs with branches are said to be more *complex* than sequential programs. Various metrics have been developed to quantify how complex a program is. For example, McCabe (1976) published the cyclomatic complexity metric.

2.5.3 Validation of Metrics

Meneely et al. (2012) acknowledge that software engineers and researchers do not have a formal set of rules for what constitutes metric validation. They conducted a systematic literature review and produced a list of forty–seven criteria identified from twenty papers that matched their criteria. We do not repeat the criteria here but these range from *a priori validation* that requires the author of the metric to hypothesise first what attribute being measured is statistically significant in explaining some phenomenon. They also require that in order for the metric to be valid it must be meaningful, in the sense that it allows for a manager or software engineer to make some decision based on the value of the metric. They refer to this as actionability. A more obvious validation criterion includes 'definition validity' which requires that the metric is stated correctly so that it can be implemented. The original mathematical definition of the LCOM1 metric failed this criteria as it always evaluated to zero.

2.5.4 Metric Tools

The MOOSE metrics suite[14] is a metrics tool to help developers evaluate and browse their code. Evaluation is supported through polymetric views including those described by Lanza et al. (2005). MOOSE is not tied to a specific programming language. Programs are represented internally using the FAMIX metamodel. Any language can be supported by providing a transformation from source code in the language to FAMIX.

McQuillan (2011) also takes a model driven approach to calculate software metrics. She developed a metamodel that describes the measurable components of a software system. This is utilised by another metamodel, which defines the actual operations for calculating metric measurements. These are annotated with OCL expressions[15] that provide the operations with semantics.

We highlight the disparity between metric tools and note that the same metric can yield different results in different tools. This is due to tool developers interpreting the metric definitions differently. This disparity has been confirmed in a study by Lincke et al. (2008).

[14]http://www.moosetechnology.org
[15]http://www.omg.org/spec/OCL/

Other than evaluating code quality, attempts have been made to use software metrics to identify where refactorings have occurred in subsequent versions of software. We discuss this interesting usage briefly next.

2.5.5 Metrics to Identify Refactorings

Refactoring detection is an area of research that involves identifying where refactorings have occurred in subsequent versions of software. The motivation for this research is to develop tools that can show developers working on large open scale projects, for example, where their changes have moved after code restructuring. A better approach would be for the refactoring engine to record and replay the changes in front of the developer, similar to what Henkel and Diwan (2005) achieve. However, their approach assumes that developers refactor with compatible refactoring engines. We know this is not the case and point out that some developers still prefer to use text editors such as Emacs[16] rather than full IDEs with refactoring support.

Schneider et al. (2010) have evaluated the potential of using software metrics to identify refactorings. Their hypothesis is that, between two classes, a small change in the Euclidean distance of fifty–four different metrics is likely to be the result of refactorings and larger changes are likely to be the result of feature additions. They evaluated their approach using three versions of the Struts web–framework. They concluded that metrics were not a promising approach for determining if a change to code was a refactoring or not.

2.5.6 Summary

The purpose of this chapter has been to examine the literature to investigate the state–of–the–art and to address the three research questions listed in section 2.1.

In response to RQ1, the literature states that in general it is not possible to prove that refactoring preserves observable behaviour for all refactorings (Mens et al., 2005). However, we have identified some successful attempts to do so for formal specification languages with well defined semantics. Less 'water tight' approaches have also been taken such as Soares et al. (2010) who resorts to random testing. We believe that a better approach (for now) would be to allow engineers using refactoring engines to alter preconditions or postconditions for refactorings easily in the field. Thus any conditions they identify as being over or under specified can be corrected immediately.

With regard to RQ2, very little effort has been made to allow developers to define their own reusable refactorings. The more well known refactoring engines such as that provided by the Eclipse IDE allow for refactorings to be composed but only manually. Larger refactorings cannot be saved for later application. Furthermore, software source code appears to get a lot of attention in refactoring circles but very little focus is placed on refactoring other artefacts such as UML diagrams. We believe that a scriptable refactoring engine that considers alternative artefacts is required to appease contemporary software engineering efforts.

In relation to RQ3 there is also very little consideration paid to overall quality while refactoring. The exception being the work of O'Keeffe and Ó Cinnéide (2006, 2008a,b) whose CODEIMP tool searches for the optimum refactoring to apply.

Clearly, there is scope for improvement with current generation refactoring tools. We believe

[16]https://www.gnu.org/software/emacs/

by taking an MDE approach to refactoring we can develop a refactoring engine that can refactor heterogeneous artefacts while paying consideration to quality as well as making it scriptable. Moreover, we believe that we can even implement such a system using MDE principles. We begin to do so in the next chapter as we develop a language to define and script refactorings.

Chapter 3

RefDstl: A Language for Specifying and Scripting Refactorings

In this chapter, we take our first stride towards developing the RefDstl system: we develop the RefDstl language for specifying and scripting refactorings. We begin with a brief discussion regarding use cases for the language and who we anticipate will use it. We discuss these issues in order to steer the language's design, which we subsequently discuss. This includes choosing a language paradigm and developing the abstract and concrete syntax. The semantics of the language is not discussed, although, it should be self explanatory. The implementation of an interpreter is described in Chapter 4 that effectively provides an operational semantics. As the chapter comes to an end, we introduce our standard library of primitive refactoring operations, written in the RefDstl language, and we also provide a small example on using the language to refactor classes from a fictitious video game.

3.1 Use Cases

We develop a language that can be used to define new refactorings through the composition of a set of primitives provided in a standard library. The language also allows for the application of the refactorings to be scripted using embedded commands that instruct the interpreter regarding what system is to be refactored.

The primary arena where we envisage this language being used is in software production environments. For example, in industry different organisations have their own coding guidelines such as that all class attributes must be self–encapsulated (i.e. access to attributes should be through the accessor and mutator methods even in the body of the class where the attributes are defined). Different developers usually have their own style of programming that can conflict with the organisation's guidelines. The developer could manually check that their code meets the guidelines but this is tedious and not a productive use of time. Instead, the organisation could write a script in RefDstl with refactorings that corrects non–conforming code (such as applying the self encapsulate field refactoring). While many refactoring engines can already perform a self encapsulate field refactoring they require the user to work through a graphical interface. Our scripting language allows for a higher degree of automation. We concede that our language is not sufficiently powerful to describe all coding guidelines. In particular, lower level guidelines that describe the structure of the text are not possible with RefDstl. For example, it would not be possible to specify that a space must occur between each operator and its operands or that chain brackets are 'lined up'.

We point the reader interested in such low level changes to the work on 'delta oriented programming' by Schaefer et al. (2010). We believe their approach could be easily extended with regular expressions that would allow these lower level changes to be performed.

We believe this language also has potential for use in academic circles. For example, consider the research area of refactoring detection. Dig et al. (2006) and others identify refactorings that have occurred between subsequent versions of software. Their motivation is to allow developers to see where their code contributions go after a program undergoes refactoring. However, the associated tools that each of these researchers develop implement different metamodels for internally describing refactorings. We view this situation as unsatisfactory. It is difficult for the outputs of the different tools to be automatically compared. It also prevents users from harnessing the power of multiple tools. They should be able to merge results to provide more accurate or complete results. The language that we present here offers a solution. Although the language is text based, which might be cumbersome to have a machine produce, it has an underlying Ecore metamodel. We suggest that researchers could programmatically instantiate that metamodel and a general model–to–text transformation be provided to transform the model into a textual RefDstl script.

Of course, nobody will use our language if it is not 'good'. We discuss the properties of 'good' programming languages next.

3.2 Characteristics of a Good Programming Language

We have identified the following properties that a programming language should have in order for it to gain acceptance. We bear these in mind in the subsequent sections as we discuss our implementation of the RefDstl language.

- Easy to learn — When faced with a new programming language, a programmer evaluates if the gains from learning how to utilise the language effectively outweight the initial learning cost. A language that is easier to learn has a higher chance of being adopted when in competition with more difficult alternatives. Clearly, we should aim to make RefDstl simple to increase its chances of adoption. RefDstl has a higher opportunity of adoption if it follows a simple design. To obtain insight into what makes one programming language easier to learn over others, we looked toward the area of computer science education. A study by Milne and Rowe (2002) involved a questionnaire of second year students in the process of learning C++ at the University of Dundee as well as academic staff teaching programming languages from around the United Kingdom. They concluded that the programming constructs that students find most difficult to understand relate to what is happening in the machine's memory during execution. Although students will probably learn to adapt by the end of their studies, we bear their conclusions in mind and suggest that the source of the problem is students having to grasp the concept of mutable state of program variables unlike variables more familiar to them from mathematics.

- Ease of use — A programming language should have a clear and consistent semantics that allows the developer to produce programs or scripts with no side affects. This is in line with guidelines developed by Karsai et al. (2009), who advocate that (domain specific) languages should be consistent and have a defined purpose; if a feature does not contribute to that purpose then it should be removed.

- Power of expression — Languages should be powerful in expression to allow programmers to succinctly describe their objective. RefDstl must at the very least be capable of describing primitive and composite refactorings and also allow describing where the refactorings are to be applied. Ideally, the language should allow for the developer to easily specify multiple locations where refactoring is to be applied. This is in contrast to graphical approaches to refactoring where the developer has to select each location for application individually.

- Tool support — Tool support for a programming language is as important as the language itself. Good tools aide developers to complete tasks quicker, which can lower development costs. For a general purpose programming language, the bare minimum that should be provided is a compiler or interpreter. This is true of some domain specific languages as well, including RefDstl. Some DSLs require tools to transform programs in the DSL to general purpose languages. For example, SQL queries can be turned into 'C' calls in order to evaluate them. In addition, tools that help to resolve programming errors by providing feedback to the programmer or help to make the code more comprehensible are also desirable. Debuggers, for example, allow the programmer to step through the program to identify points in the program where unexpected behaviour occurs. The development of a debugger for RefDstl is beyond the scope of this work. However, we do provide an editor that offers syntax highlighting and content assist in the Eclipse IDE as well as an execution environment.

3.3 Design of a DSTL for Defining and Scripting Refactorings

The following cascading stages form a framework for designing and implementing new programming languages. We address each of these in turn while discussing the creation of the RefDstl language in the subsections that follow.

1. Choosing a suitable paradigm for the language — The paradigm of a language defines its high level design principles. For example, does the programming language place emphasis on objects or actions? The chosen paradigm influences how the language is expressed in text, i.e. its syntax. For example, if the language permits objects then the language must allow a way of defining/instantitating objects.

2. Defining a syntax of the language — The syntax defines the structure of the language. For example, what are the keywords of the language and what order must they be composed to form valid 'sentences'. The syntax of a language is expressed as a grammar that consists of production rules. The grammar file is then input to a parser generator, which produces a parser. The parser is essentially a language recogniser: It accepts only those inputs that conform to the grammar specification. The syntax chosen is naturally dependent on the constructs available in the language.

3. Specifying a semantics of the language — The elements of the language's syntax must have an explicit meaning. For example, a common question arises in language design

when invoking methods/procedures/functions in the language, that is, how are the arguments passed? They could be passed by value or by reference, for example. The language might even offer syntax that allows the programmer to switch between the two.

4. Implementing an execution environment — Once the semantics of the language have been defined then software for interpreting the language must be developed. Alternatives include producing compilers, which transform the source code program into an executable that is usually a binary executed directly on the machine or an intermediate file that is executed on a virtual machine, which is the case with Java or the .NET family of languages.

3.3.1 Choosing a Suitable Paradigm

There are various programming paradigms in existence such as imperative/declarative or object–oriented/procedural. Programming languages can even fall into multiple paradigms.

- Imperative — These programming languages usually include low level constructs that allow the programmer to explicitly control the behaviour of the underlying hardware. For example, they might have the concepts of variables and assignments to manipulate the state of the machine, branching statements that allow the machine to move from one state to another when certain conditions are met, or iterative constructs that move the machine through a sequence of states repeatedly until other conditions are met. The languages that find most use today are imperative in nature, such as ‘C’, C++, and Java.

- Declarative — These are higher level languages and programmers specify *what* they want done rather than *how* they want it done. SQL is, perhaps, the most widely used declarative programming language that is used for data definition and manipulation in relational database systems. Declarative languages can be quicker to program with and often require fewer lines of code. This is especially true when designed for domain specific purposes, such as the case with SQL. Consider how an SQL query that joins two tables would need to be written in ‘C’ with a loop that populates some data structure with the Cartesian product of the two tables followed by a loop that iterates through the structure to remove rows where the appropriate columns do not match under conditions. General purpose declarative programming languages also exist such as Haskell. The productivity gains that come with using declarative languages often come at the cost of expressibility. For example, often developers will find it easier to describe an algorithm imperatively using variables (rather than bindings that appear in declarative languages such as Haskell), loops and if statements.

- Object–Oriented — In the Object–Oriented (OO) paradigm, the focus of development is on writing reusable modules of code called classes. A class is a container for data which is stored in variables called fields (or attributes) and methods which operate on that data. A good class is highly cohesive, i.e. each method in the class reads from or writes to a high proportion of the class’ fields. It is also loosely coupled: it depends very little on other classes and allows other classes to depend very little on it.

- Procedural — Procedural programming places the focus of development on writing procedures of code, which are grouped lists of instructions. Procedural languages heavily contrast with OO languages in that the procedural languages place emphasis on actions to be done rather than on the data.

In choosing a paradigm for RefDstl, we decided to take a balanced approach by providing a language that allows the programmer to declaratively describe what refactorings to apply but enriching it with a 'for' looping construct to permit repetition in the language and a conditional 'if' construct. It would have been possible for us to provide a 'map' and 'filter' function, which are suitable alternatives and more the 'norm' in declarative languages but our choice is driven by what we believe is most familiar to the target end users. It would be possible to implement different variants of RefDstl following different paradigms so that a user could choose the paradigm most familiar to them. Our reliance on MDE technology (discussed in section 3.4) to implement RefDstl means that the same interpreter could have been used to interpret scripts written in all of the variants by performing model–to–model transformations to some 'baseline' paradigm.

3.3.2 Language Constructs

We discuss the constructs provided by RefDstl in this section, which allows us to develop an appropriate syntax later in section 3.4.

- Modularisation via libraries of procedures — For the obvious benefit of code re–use, we implemented the concept of 'libraries' into the language. Each script is itself a module that is uniquely identifiable based on its location in the file system. This path is the script's identifier that can be used to import it. Unlike other programming languages that include visibility control features (such as private methods in Java) that permit for elements of a module to marked as hidden to external modules, all elements of a script are visible to any other script importing it. A script is imported by using the 'imports' keyword at the beginning of the script. Multiple scripts can be imported by writing imports statements one line after the other.

- Modularisation of scripts via procedures — We believe in the guidelines of structured program that advocate structuring code into blocks called procedures that perform one well defined task. RefDstl supports the notion of procedures for this purpose. Procedures are exactly that; they are not functions. A procedure never returns a value.

- Preconditions and Postconditions — Following the practices of design by contract, RefDstl allows for procedures to be annotated with pre– and post– conditions, which are expressions that can reason about the program being refactored in its representation as a MoDisco model. Plugins can be developed for RefDstl that provide hooks (which we call script handles), which further enhances what can be reasoned about in these conditions. We have incorporated a plugin with a hook that allows the conditions to reason about the program represented as an instance of the metrics metamodel developed by McQuillan (2011).

  These conditions serve a dual purpose. The first is that they ensure the refactoring primitive operations work correctly. It might seem unusual to ensure the correctness of primitive operations in a script rather than in the transformation engine but this provides a benefit. As agreed by Overbey and Johnson (2011), the formulation of refactoring conditions is actually the most difficult part of developing a refactoring engine.

By separating the conditions from the code, it makes it convenient to debug overly strong preconditions (as they can be commented out). A standard library of refactorings is provided with the RefDstl system so the end user will be unaware of this. The second purpose is that it allows RefDstl end users to define their own refactorings with their own conditions, that can be used to verify the correctness of the refactorings with respect to properties that they specify.

Unlike some implementations of DBC, RefDstl does not provide syntax that would allow for the total correctness of the program to be verified. It only permits checking for 'partial correctness'. This means that we are limited to verifying *if* the program completes then the conditions are guaranteed to have held. We make no guarantee that the script will finish executing. To add total correctness to the language, we would need to include loop variants[1]. Pre and postconditions are simple boolean formulae built using the familiar boolean connectives as well as existential or universal quantifiers.

- Procedure Invocation — Within the body of a procedure, the user must have a way to invoke other procedures and pass parameters. We provide the 'call' keyword for this purpose.

- Branching — The language provides branching constructs to the user. We limit this to 'if' statements without 'else' blocks. 'Else' blocks could be easily added at a later point but during the development of the standard library of refactorings (in Appendix B) their inclusion was found to be unnecessary. The language currently includes the necessary boolean operations to work around this if it poses difficulties to the user.

  For the sake of maintaining consistency in the language, we permit the same format in 'if' conditions for describing guards as is used in the formulae in pre and post conditions. This means that guards in RefDstl can be more expressive than guards in other languages, such as Java 7, which do not permit quantification in conditional guards.

- Iteration — The language also provides an iteration construct for looping over collections of primitive refactoring instructions.

- Refactoring primitives — These statements manipulate the program under refactoring, such as renaming fields, methods etc.

3.4 RefDstl Implementation

Up until now when discussing the language's syntax we have been referring to the 'concrete' syntax, i.e. what the programmer types as a program. This is too high level and cumbersome to translate by compilers or understand by interpreters. So an alternative, simpler view of the syntax is used. This is known as the abstract syntax, which Völter et al. (2013) refers to as a 'data structure that holds the core information of the program'.[2] The abstract syntax forgets the insignificant parts of the program. For example, it drops characters that are used as delimiters such as whitespace (to

[1] In contrast to loop invariants that describe a property of the program's state that is true upon each iteration of the loop, a loop variant is a value that must decrease upon each iteration.

[2] We believe the data structure is a representation of the abstract syntax and not the abstract syntax itself but that distinction bares no consequence here.

separate terms), semi–colons (often used to separate statements) or brackets (to create precedence in complex expressions). These can be omitted without losing the meaning of the program because they only serve as hints to the lexer (which separates the terms in the program text) and the parser as to what parts of the program have a higher level of precedence. The output from a parser is a parse tree with typed nodes. Interpreting or compiling the program involves traversing the tree to generate machine code instructions or interpreting the nodes of the tree and performing some action on a virtual machine (as is done by an interpreter).

In this section, we discuss our use of the Xtext language framework for developing RefDstl.

3.4.1 Parser Generators and Grammars

A parser generator is a tool that takes as input a specification of the concrete syntax and produces as output the code for a parser. When compiled this can be used to parse programs that fully meet the language specification. Moreover, if the parser is provided with input that does not conform to the specification of the language then it will halt before it finishes parsing the program. 'User friendly' parsers also provide feedback to help the programmer to locate the problem with their program such as line number or character the parser realised a problem had occurred.

With traditional parser generators, like JavaCC[3], it is the language designer's responsibility to write the classes that represent the abstract syntax. He or she must also embed code into the concrete language specification that builds up the abstract syntax tree as the parser reads through the program text. This is done by placing code snippets inside the grammar specification.

More recent language frameworks have been developed that make it easier to develop languages (specifically DSLs). These make it easier and less time consuming to develop languages because they automatically derive the abstract syntax based on the grammar of the language.

3.4.2 A Grammar for RefDstl

Grammars specify the syntax of programming languages. They are regularly provided in Extended Backus–Naur form (EBNF). In Backus Naur Form (BNF), a grammar consists of these components (Aho et al., 2006):

- Terminal symbols — These are keywords and other symbols in the language such as symbols for built in operators, brackets and braces for describing lists or grouping statement or symbols to denote separation of statements such as semi–colons.

- Non–terminals — A non–terminal is a variable in the grammar that is associated to a production rule. There must be exactly one non–terminal to mark the beginning of the grammar.

- Production rules — A production rule is a mapping from a non–terminal to a sequence of terminals or non–terminals. The mapping is non–deterministic as a non–terminal can be mapped to more than one sequence. The rule is divided into a head and a body. What is on the left of the rule is known as the head and the body is on the right.

EBNF extends BNF by allowing elements of regular expressions to appear in production rules. This does not make BNF more powerful but it is a convenience.

[3]https://javacc.java.net

The full grammar for the RefDstl language is presented in Appendix A. We used this grammar to generate a parser and metamodel for the RefDstl language with Xtext. Auxiliary tools such as a plug–in editor for Eclipse were also produced.

3.4.3 Substituting RefDstl

There is an additional and somewhat subtle benefit from using the Xtext toolset to develop our language. That is to say that when an Xtext grammar file is compiled it produces an Ecore metamodel. The parser that Xtext also produces builds instances of this metamodel that represent the parsed language. However, the metamodel is readily available for use by tools. We propose that researchers and developers who work on tools that involve building in–memory models of refactorings or indeed serialising them to a file should use our metamodel. This allows those researchers and developers to take advantage of some of MDE's main benefits, i.e. tool re–use and portability. There is a downside however to instantiating the metamodel programmatically, i.e. the metamodel can be instantiated in a way that would never be possible using text input to the generated parser.

3.5 Defining a Standard Library of Java Refactorings

We have thus far described the goals and design guidelines for RefDstl. With those goals in mind we described a list of constructs required and specified the syntax of the language. We now provide a standard library of refactorings, which we use in the sample scripts used in the following section. The code that makes up the standard library is presented in Appendix B. It shows how to use the language, in particular how to make procedure calls and write preconditions over the model representing the program being refactored. All of the primitive refactoring instructions that the language offers are covered in the library; they are in the bodies of the numerous procedures.

On the surface, it might seem unusual that we wrap the instructions of our language inside procedures that we encourage others to use. The reason for this is that the instructions do not have any preconditions associated with them. The interpreter will execute the instructions regardless of the state of the program being refactored. The only way in RefDstl to annotate the instructions with preconditions is by placing them inside procedures as we have done. Users of RefDstl can freely replace our library with their own if they find ours to be insufficient for their needs. It is worth noting that our library does not come with any postconditions. These can be included by either editing the library itself or by extending it. We omitted postconditions as we were trying to adhere to the primitive refactorings given by Fowler (1999).

3.6 A Small Case Study

In this section, we use RefDstl to repeat a small case study performed by O'Keeffe and Ó Cinnéide (2003).

A UML class diagram for the original design of a fictitious video game is shown in Figure 3.1. At the top of the class hierarchy is the `Weapon` class with a private attribute called `power` that is accessible to the five child classes via the public `getPower` method. Its immediate children, `MissileWeapon` and `MeleeWeapon`, both have an attribute called `range`. We might say that these two child classes exhibit the 'duplicate code smell'. The re–course here would be to 'pull up' the `range` field and the `getRange` method to the `Weapon` class to aim for the design presented in Figure 3.2. O'Keeffe and Ó Cinnéide (2003) call this maximising inheritance.

The RefDstl script for performing this refactoring is listed in Listing 3.1. At the beginning of the script, we import the standard library of refactoring primitives, which were discussed in the previous section. The next two lines instruct the interpreter about what model is to be refactored and where the resulting MoDisco model of the refactored system is to be persisted. The declaration of the `main` procedure follows. This is where the script begins its execution. It must be at the top of the script following the input/output directives. In this script, the main procedure delegates its task to a procedure called `safeRefactor`. This delegation may seem wasteful but the `main` procedure cannot be annotated with pre or post conditions so placing the refactoring instructions inside another procedure allows us to work around this. Future versions will remove this restriction. Note that in the RefDstl scripts that the tilde symbol is used in postconditions to mean 'the version of this class prior to the refactoring'.

The definition of `safeRefactor` declares six preconditions identifiable by the `requires` keyword. All of these preconditions serve a similar purpose: to ensure that only existing classes are refactored. If any of the classes do not exist then the interpreter will halt executing the script. The preconditions take advantage of the `$utils` script handle that has a method provided named `classExists`. It returns a boolean value to indicate if a class by a particular fully qualified name is declared in the project undergoing refactoring. Other, more interesting, preconditions are annotated on the procedures that are defined in the standard library of refactorings (see Appendix B) and used in this script. Those preconditions are not repeated in Listing 3.1 since the purpose of the standard library is to minimise repetition of preconditions.

The definition of `safeRefactor` declares twelve postconditions that are indicated by the `ensures` keyword. These reason about the available methods and methods inherited metrics, which are calculated using the `am` and `mi` operations from the McQuillan (2011) metrics metamodel that is accessed using the `$metrics` script handle. The first postcondition[4] states that the `Weapon` class will inherit the same amount of methods when the `getRange` method is pulled up. Following this, we state that `MissileWeapon` and `MeleeWeapon` will inherit more methods after the refactoring. This is to be expected since we are pulling a method from each of these classes and locating it in the `Weapon` class. The classes `Bow`, `Javelin` and `Sword` continue to inherit the same number of methods. We then reason about the available methods. Naturally the `Weapon` class will have more methods available after the refactoring because the `getRange` method will be then declared in it. The classes that inherit from `Weapon` either directly or indirectly will have the same number of methods available after the refactoring since they will be able to access the `getRange` method from their parent classes.

```
imports "standard_library.refdsl"

in "model_original/model_java2kdm.xmi"
out "max_inheritance/model_java2kdm.xmi"

main() {
  call safeRefactor()
}

/**
```

[4]A discussion covering how these are implemented, in particular 'old' variables, is reserved until subsection 4.2.5.

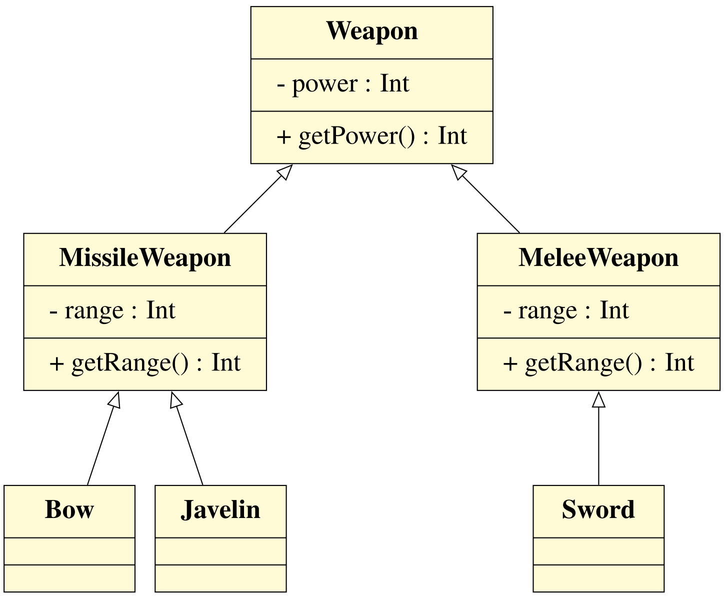

Figure 3.1: A UML class diagram demonstrating a poorly designed system to be refactored.

```
 * Performs a pull up refactoring on the getRange procedure
 * and range field.
 *
 * It also ensures that the metric values are changed appropriately.
 */
proc safeRefactor()
 requires $utils.classExists("Weapon")
 requires $utils.classExists("MissileWeapon")
 requires $utils.classExists("MeleeWeapon")
 requires $utils.classExists("Bow")
 requires $utils.classExists("Javelin")
 requires $utils.classExists("Sword")
 ensures $metrics.mi($utils.findClass("Weapon")) == $metrics.mi($utils.
    findClass("~Weapon"))
 ensures $metrics.mi($utils.findClass("MissileWeapon")) > $metrics.mi($utils.
    findClass("~MissileWeapon"))
 ensures $metrics.mi($utils.findClass("MeleeWeapon")) > $metrics.mi($utils.
    findClass("~MeleeWeapon"))
 ensures $metrics.mi($utils.findClass("Bow")) == $metrics.mi($utils.findClass("
    ~Bow"))
 ensures $metrics.mi($utils.findClass("Javelin")) == $metrics.mi($utils.
    findClass("~Javelin"))
 ensures $metrics.mi($utils.findClass("Sword")) == $metrics.mi($utils.findClass
    ("~Sword"))
 ensures $metrics.am($utils.findClass("Weapon")) > $metrics.am($utils.findClass
    ("~Weapon"))
 ensures $metrics.am($utils.findClass("MissileWeapon")) == $metrics.am($utils.
    findClass("~MissileWeapon"))
 ensures $metrics.am($utils.findClass("MeleeWeapon")) == $metrics.am($utils.
    findClass("~MeleeWeapon"))
 ensures $metrics.am($utils.findClass("Bow")) == $metrics.am($utils.findClass("
    ~Bow"))
 ensures $metrics.am($utils.findClass("Javelin")) == $metrics.am($utils.
    findClass("~Javelin"))
 ensures $metrics.am($utils.findClass("Sword")) == $metrics.am($utils.findClass
    ("~Sword"))
{
  call refactorField()
  call refactorMethod()
}

/**
 * This performs the pull up field refactoring on the range field
 * range from classes MissileWeapon and MeleeWeapopn.
 */
proc refactorField() {
  call moveFieldToSuperclass(range, MissileWeapon)
  call deleteField(range, MeleeWeapon)
}

/**
 * This pulls up the getRange method from MissileWeapon and MeleeWeapon.
 */
proc refactorMethod() {
  call moveMethodToSuperclass(getRange, MissileWeapon)
  call deleteMethod(getRange, MeleeWeapon)
}

```

Listing 3.1: A RefDstl script that performs the pull up refactoring on the `range` field and `getRange` method for the `MissileWeapon` and `MeleeWeapon` classes presented in Figure 3.1.

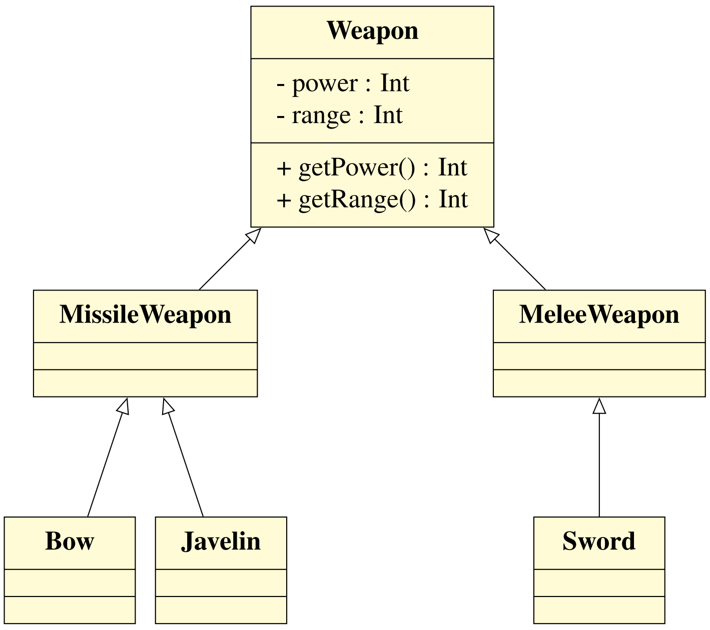

Figure 3.2: A UML class diagram that represents the system presented in Figure 3.1 refactored to obtain maximum inheritance.

This design is not much better. It means every class has a larger than necessary interface. Software engineers generally try to decrease the size of the interface to reduce the risk of afferent coupling. It provides them with leverage to be able to eliminate one or more of the methods without having to consider the ramifications for client code. To minimise the interface offered by the classes we aim for the design presented in Figure 3.3 and provide a RefDstl script for doing so in Listing 3.2. The structure of this script differs to Listing 3.1 in that we now use the `for` construct in the definition of the `buryMethod` procedure. It repeatedly applies the push down refactoring to send an entity to the lowest level of a class hierarchy.

```
imports "standard_library.refdsl"

in "model_original/model_java2kdm.xmi"
out "min_methods/model_java2kdm.xmi"

main() {
 call safeRefactor()
}

/**
 * Safely pulls up the range field into the Weapon class and pushes down
 * the getPower and getRange methods.
 *
 * It also ensures that the metric measurements for the available methods and
    methods inherited
 * measurements update appropriately.
 */
proc safeRefactor()
 requires $utils.classExists("Weapon")
 requires $utils.classExists("MissileWeapon")
 requires $utils.classExists("MeleeWeapon")
 requires $utils.classExists("Bow")
 requires $utils.classExists("Javelin")
 requires $utils.classExists("Sword")
 ensures $metrics.mi($utils.findClass("Weapon")) == $metrics.mi($utils.
    findClass("~Weapon"))
 ensures $metrics.mi($utils.findClass("MissileWeapon")) < $metrics.mi($utils.
    findClass("~MissileWeappn"))
 ensures $metrics.mi($utils.findClass("MeleeWeapon")) < $metrics.mi($utils.
    findClass("~MeleeWeapon"))
 ensures $metrics.mi($utils.findClass("Bow")) < $metrics.mi($utils.findClass("~
    Bow"))
 ensures $metrics.mi($utils.findClass("Javelin")) < $metrics.mi($utils.
    findClass("~Javelin"))
 ensures $metrics.mi($utils.findClass("Sword")) < $metrics.mi($utils.findClass(
    "~Sword"))
 ensures $metrics.am($utils.findClass("Weapon")) < $metrics.am($utils.findClass
    ("~Weapon"))
 ensures $metrics.am($utils.findClass("MissileWeapon")) < $metrics.am($utils.
    findClass("~MissileWeappn"))
 ensures $metrics.am($utils.findClass("MeleeWeapon")) < $metrics.am($utils.
    findClass("~MeleeWeapon"))
 ensures $metrics.am($utils.findClass("Bow")) == $metrics.am($utils.findClass("
    ~Bow"))
 ensures $metrics.am($utils.findClass("Javelin")) == $metrics.am($utils.
    findClass("~Javelin"))
 ensures $metrics.am($utils.findClass("Sword")) == $metrics.am($utils.findClass
    ("~Sword"))
{
 call refactorField()
 call refactorMethod()
}

/**
 * This performs the pull up field refactoring on the field:
 * range from classes MissileWeapon and MeleeWeapopn.
 */
proc refactorField()

{
 call moveFieldToSuperclass(range, MissileWeapon)
 call deleteField(range, MeleeWeapon)
}

/**
 * The getPower method will be moved to the lowest levels in
 * the inheritance hierarchy.
 */
proc refactorMethod() {
 call buryMethod(getPower, Weapon)
 call buryMethod(getRange, MissileWeapon)
 call buryMethod(getRange, MeleeWeapon)
}

/**
 * The bury method refactoring sends a method to the
 * lowest classes in the inheritance hierarchy.
 */
proc buryMethod($methodName, $class) {
 if($class.isParent()) {
  call moveMethodToSubclass($methodName, $class)

  for($cls in $class.children()) {
   call buryMethod($methodName, $cls)
  }
 }
}

```

Listing 3.2: A RefDstl script that performs a pull up refactoring on the range field and a push down refactoring on the getPower method until it is at the lowest level of the inheritance hierarchy.

Although this is an improvement, a problem remains. In this design, `Sword` now has access

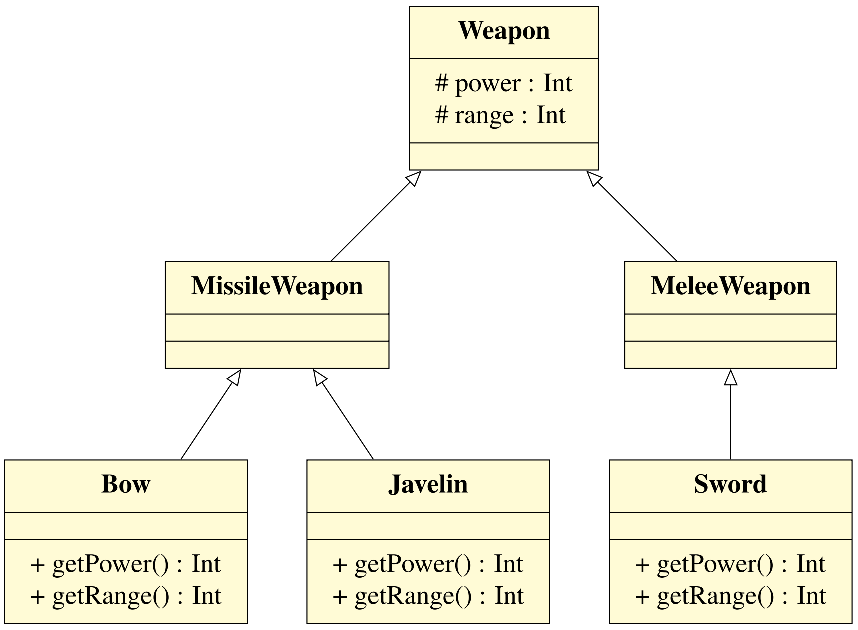

Figure 3.3: The UML class diagram from Figure 3.1 refactored for minimum methods per class.

to a method called `getRange`. A `MeleeWeapon` should not have a concept of range. We perform a final refactoring on this design to correct this and aim for the design shown in Figure 3.4, which is yielded by the RefDstl script in Listing 3.3. This is the optimised design according to search based approach to refactoring by O'Keeffe and Ó Cinnéide (2003).

```
imports "standard_library.refdsl"

in "model_original/model_java2kdm.xmi"
out "optimised/model_java2kdm.xmi"

main() {
 call safeRefactor()
}

/**
 * Safely pulls up the range field and removes the getRange method
 * from the MeleeWeapon class.
 *
 * The metric measurements are also check to ensure they are updated
    appropriately.
 */
proc safeRefactor()
 requires $utils.classExists("Weapon")
 requires $utils.classExists("MissileWeapon")
 requires $utils.classExists("MeleeWeapon")
 requires $utils.classExists("Bow")
 requires $utils.classExists("Javelin")
 requires $utils.classExists("Sword")
 ensures $metrics.mi($utils.findClass("Weapon")) == $metrics.mi($utils.
    findClass("~Weapon"))
 ensures $metrics.mi($utils.findClass("MissileWeapon")) == $metrics.mi($utils.
    findClass("~MissileWeappn"))
 ensures $metrics.mi($utils.findClass("MeleeWeapon")) == $metrics.mi($utils.
    findClass("~MeleeWeapon"))
 ensures $metrics.mi($utils.findClass("Bow")) == $metrics.mi($utils.findClass("
    ~Bow"))
 ensures $metrics.mi($utils.findClass("Javelin")) == $metrics.mi($utils.
    findClass("~Javelin"))
 ensures $metrics.mi($utils.findClass("Sword")) < $metrics.mi($utils.findClass(
    "~Sword"))
 ensures $metrics.am($utils.findClass("Weapon")) == $metrics.am($utils.
    findClass("~Weapon"))
 ensures $metrics.am($utils.findClass("MissileWeapon")) == $metrics.am($utils.
    findClass("~MissileWeappn"))
 ensures $metrics.am($utils.findClass("MeleeWeapon")) < $metrics.am($utils.
    findClass("~MeleeWeapon"))
 ensures $metrics.am($utils.findClass("Bow")) == $metrics.am($utils.findClass("
    ~Bow"))
 ensures $metrics.am($utils.findClass("Javelin")) == $metrics.am($utils.
    findClass("~Javelin"))
 ensures $metrics.am($utils.findClass("Sword")) < $metrics.am($utils.findClass(
    "~Sword"))
{
 call refactorField()
 call refactorMethod()
}

/**
 * This performs the pull up field refactoring on the field:
 * range from classes MissileWeapon and MeleeWeapopn.
 */
proc refactorField() {
 call moveFieldToSuperclass(range, MissileWeapon)
 call deleteField(range, MeleeWeapon)
}

/**
 * This will remove the getRange method from the MeleeWeapon
 * class.
 */
proc refactorMethod() {
 call deleteMethod(getRange, MeleeWeapon)
}

```

Listing 3.3: A RefDstl script that performs a pull up refactoring on the range field and a push down refactoring on the getPower method until it is at the lowest level of the inheritance hierarchy.

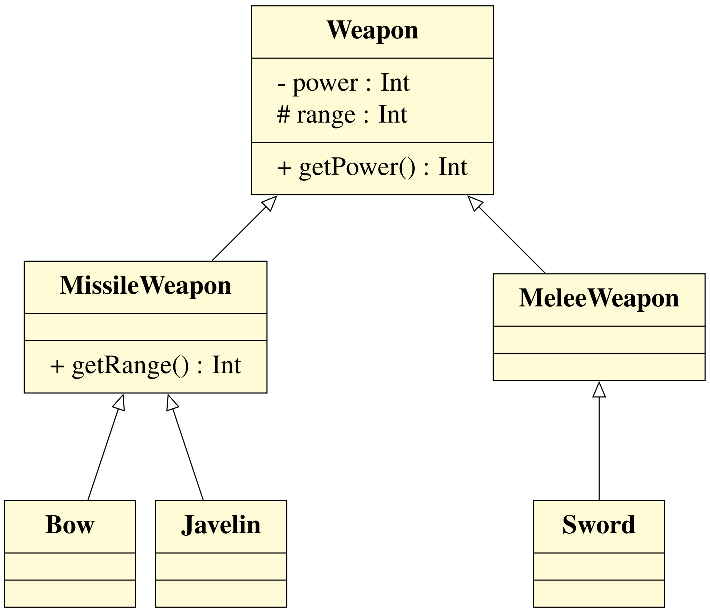

Figure 3.4: The UML class diagram from Figure 3.1 refactored using the optimisations calculated by the Dearthóir tool.

3.7 Summary

This chapter described a language to define and script software refactorings. The language was built using the Xtext language framework. It not only produced supporting tools for the language, such as a parser and Eclipse IDE integration, but it also provided a metamodel that can be instantiated by other researchers working in the area of refactoring. This metamodel allows for compatibility between tools. We also provided a standard library of primitive refactoring operations. These are used in our evaluation of RefDstl in Chapter 5. In the following chapter we discuss implementing an interpreter for the language in the form of an extensible framework that can refactor various kinds of software artefacts beyond the usual limits of source code.

Chapter 4

A Model Driven Framework to Interpret RefDstl Scripts

In the previous chapter, we discussed the syntax and implementation of the RefDstl language for specifying and scripting refactorings. We now focus on developing a framework for interpreting those scripts. We begin by describing its high level design and identify the various components that collectively form its architecture. In the latter part of this chapter, we describe how the framework can be extended with additional model transformations so that it is relevant for refactoring many heterogeneous artefacts rather than just source code. We also describe how the RefDstl language can be enriched with the notion of 'script handles'.

4.1 High–Level Overview of the RefDstl Interpreter Framework

Figure 4.1 presents a UML component diagram illustrating the main components of the RefDstl framework. We have omitted the classes inside these components from the diagram for the sake of simplicity. We describe the general purpose of each of these components and how they interact before giving a more detailed discussion about each in turn.

We saw in Chapter 3 that the RefDstl language permits modularity by allowing scripts to be split into multiple files. However, before a RefDstl script can be executed those files must be 'linked' together so that procedures defined in external libraries, for example, can be resolved. The `ScriptManager` component facilitates this need. Once a script has been linked then execution is ready to commence. The `ScriptManager` relies on the `InstructionInterpreter` interface provided by the `Controllers` component to initiate execution.

The `Controllers` component contains multiple classes, which are each responsible for interpreting specific constructs and instructions provided by the RefDstl language. However, the controllers delegate most of their work. For example, the controller responsible for invoking procedures delegates the task of maintaining the environment (i.e. the mapping of values to variables) to the `Environment` component, which it accesses through the `Stack` interface. Furthermore, the task of interpreting the pre and post conditions on procedures is delegated to the `ConditionEvaluator` component through the `Interpreter` interface. The `ConditionEvaluator` component evaluates the truthness of boolean formulae. The 'heavy lifting' required to perform model transformations is delegated to to the `Transformations` component, which is accessed via the `CompositeTransformation` interface. The `Transformations` component contains multiple classes, each of which understand how to load, transform and persist different (or heterogeneous) model types. The `ScriptHandles` component is also called upon by classes in the `Controllers` component. It provides utility methods that can be used in RefDstl scripts to enrich the language with features that are otherwise unavailable, for example, outputting messages to log files or the console.

We now elaborate on the details of each of these components.

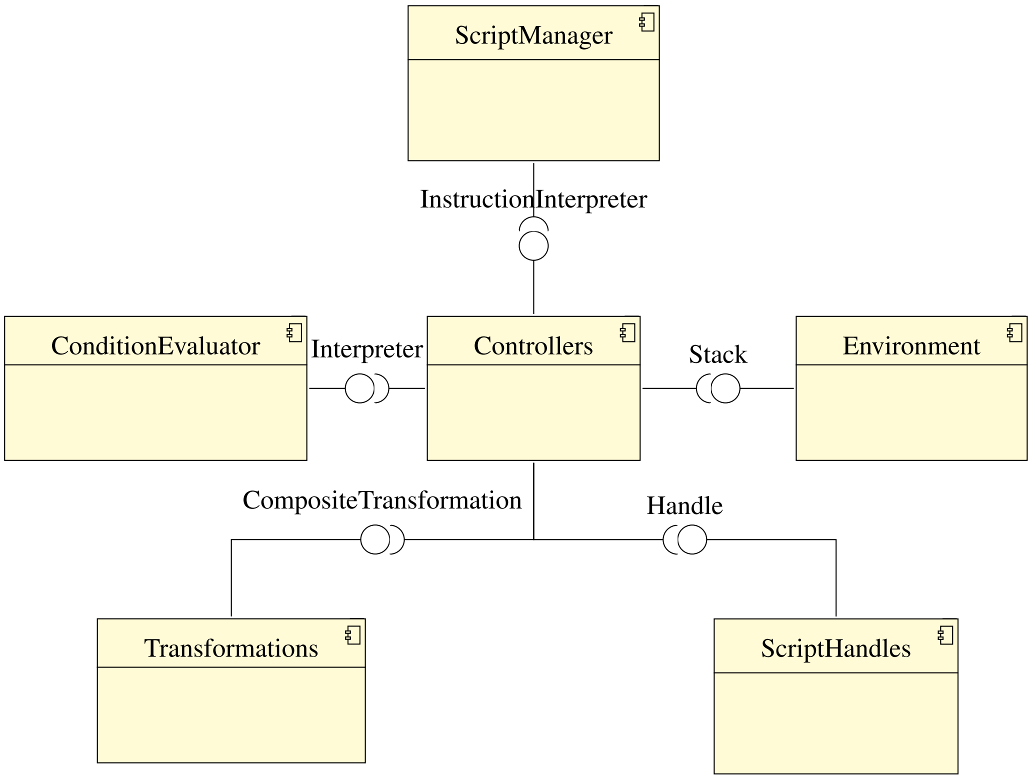

Figure 4.1: An overview of the components in the RefDstl interpreter framework.

Script Manager

The script manager has a similar purpose to a ‘linker’ used with traditional compiled software. It loads RefDstl scripts and inspects them for `import` statements. For example, the code snippet in Listing 4.1 shows a RefDstl script importing the code from a library called `my_refactorings.refdsl`. An import is equivalent to taking the procedures defined in the imported script and appending them to the bottom of the importing script. Of course an imported script, such as `my_refactorings.refdsl`, might also have its own import statements. These are recursively resolved until no more imports remain. Circular dependencies between imports should be avoided and the current implementation of RefDstl is unable to detect these. In a situation where one script and a subsequently imported script share a procedure with the same name then the procedure that was loaded the earliest is used. Parameter types are not used as part of the procedure's signature; thus, overloaded procedures are unsupported and will result in a runtime error.

```
imports "my_refactorings.refdsl"

in "some_model.xmi"
```

```
out "some_model_refactored.xmi"

/**
 * Script begins execution in main.
 */
main() {
...
}
```

Listing 4.1: An example showing the syntax used to import one RefDstl script into another.

Controllers

Once a script has been fully loaded, it begins to execute. For each of the different refactoring instructions (create class, move field etc.) we implemented a controller. Controllers also exist that are specific to aspects of the RefDstl language, such as procedure calls.

The controllers responsible for interpreting refactoring instructions are mundane delegates. They merely extract the parameters for the refactoring instruction (such as the new name for a class in a 'create class' instruction) from the script and resolve any variables to their values using the `Environment` component. They then pass the parameters to the appropriate method in `CompositeTransformation` singleton object in the `Transformations` component. This repeats the message to all of the transformation classes registered in the framework, such as the transformation for refactoring the MoDisco model of the system or the transformation responsible for the metrics model of the system.

Other controllers that are specific to RefDstl language constructs have more interesting and involved activities. For example, the `MainInt` class in the `Controllers` component is responsible for beginning execution of the script. It attempts to locate the `main` procedure in the refactoring script and begins to execute the list of instructions in the body. Since a controller only knows how to interpret one type of instruction, it dispatches everything that it does not understand to the general instruction controller, named `InstructionInt`. This controller inspects the name of the actual type of the instruction and by using Java's reflection capabilities it instantiates a new instance of the correct controller to perform the interpretation and further delegation occurs.

The `ConditionalStatementInt` is responsible for interpreting conditional `if` statements in the language. Guards on conditional statements use the same syntax as expressions in pre and post conditions so the interpretation of the guard is delegated to the conditions evaluator (discussed below). If the guard evaluates to true then `ConditionalStatementInt` evaluates the body of the condition through further use of delegation. Otherwise, it does nothing. For example, Listing 4.2 presents a procedure that will rename a class declaration to `Bar` if its name is currently `Foo`. The procedure takes as input a class declaration and the conditional statement inside the body of the procedure uses a guard to determine if the name of the passed class declaration is named `Foo` (in which case the guard evaluates to boolean true). If the guard is true then `$possibleFoo` will be renamed otherwise no action is taken.

```
/**
 * This method will rename a class called Foo to Bar.
 * Otherwise it does nothing.
 */
```

```
renameFooToBar($possibleFoo) {
    if($possibleFoo.getName().equals(Foo)) {
        call renameClass($possibleFoo, Bar)
    }
}
```

Listing 4.2: An example showing the syntax used to describe a conditional statement in RefDstl.

The `LoopingStatementInt` is responsible for performing a sequence of instructions over a collection. It extracts the object expression from the loop statement header. It delegates the task of interpreting this to the `ObjectExpressionInt`. The `ObjectExpressionInt` returns the result of the expression, which the `LoopingStatementInt` casts to an instance of a collection. Trying to loop over anything other than a collection is an obvious error. The `LoopingStatementInt` iterates over each member of the collection. During each iteration, it maps the current element to a variable name (determined by the script) in the environment (described later) and then executes each instruction in the loop body. In Listing 4.3, for example, we iterate over all of the classes in the project under refactoring and create an instance variable for logging. The `ObjectExpressionInt` evaluates `$utils.allClasses()` to a collection and the `LoopingStatementInt` binds each member of this collection to the variable named `$cls` during each iteration. The body of the loop is applied to the bound variable during each iteration, which in this case creates the variable for logging.

```
/**
 * This loop will add a variable for logging to each class.
 */
for($cls in $utils.allClasses()) {
 call createField(LOG, public, $cls.getName(), "java.util.Logger")
}
```

Listing 4.3: An example showing how to iterate in RefDstl.

The `ObjectExpressionInt` evaluates object expressions. These begin with a variable followed by a cascade of method calls. The parameters to these method calls can also be other object expressions. For example, in Listing 4.4 there are two object expressions that we use to compare two class names for equality. The first object expression to be executed is the nested expression `$clsTwo.getName()`. The `ObjectExpressionInt` locates the variable `$clsTwo` from the environment and then using Java's reflection mechanism invokes the `getName()` method. The resulting object is then cached. Next, `ObjectExpressionInt` locates the object assigned to the variable `$clsOne` and invokes the `getName()` method on it before, finally, evaluating the cascaded method `equals(...)`. The earlier cached result is passed as a parameter. The result of all object expressions is an object; the actual type such as `String` is unneeded.

```
$clsOne.getName().equals($clsTwo.getName())
```

Listing 4.4: An example object expression in RefDstl.

`ProcedureCallInt` handles RefDstl `call` instructions. In the case where no arguments are being passed with the procedure call this is simply a case of searching the script for the procedure with the same name and then proceeding to have `Controller` interpret each of the statements in the procedure's body in turn. However, prior to executing the body `ProcedureCallInt` requests that the environment create a new frame on the stack, which is later destroyed when the procedure is finished executing. The purpose of the frame is to provide scope to the variables. Without frames the instructions in the body of a procedure would be able to read the parameters of the invoking procedure. This could lead to unusual behaviour. Once the procedure is finished executing the frame is destroyed for efficient memory usage. The `ProcedureCallInt` class is also responsible for placing the arguments on the stack frame. This is achieved by getting the names of the *parameters* that the procedure being called expects and the variable names or values of the *arguments* that are being provided. If the size of the argument list does not match the parameter list then an error is raised. Otherwise, on the frame created for the procedure call, each i^{th} argument is mapped to each i^{th} variable name. The procedure's body is then executed using, once again, more delegation.

Transformations

Model transformations are the primary motivation for this framework. There are no technical restrictions on the kind of models that can be transformed. However, the framework is intended to be used with transformations that are driven by refactoring instructions. The actual act of transforming the model is not required to be done via Java as we have done in our provided transformations. Developers extending the framework can choose to have the transformation conducted by calling out to an ATL script, for example. We believe this flexibility is a benefit of our framework.

We provide two transformations with the framework. The first is `MoDiscoTransformation`. If the framework is equipped with just this transformation then it behaves as any regular refactoring engine. Its responsibility is to transform MoDisco models that are representations of the Java system being refactored. It also provides a handle named `$project` in scripts. This can be used in the same way as variables in the RefDstl language. We do not go into detail about how this transformation actually refactors models. However, we point out a distinguishing feature that distinguishes the framework from other refactoring engines. The preconditions that most refactoring engines would check prior to the transformation are not embedded in the Java code that performs the transformations. Instead, the transformation assumes that the conditions have already been verified prior to the request to perform the transformation. In other words, the conditions must be expressed in the refactoring script in pre and post conditions on procedures. This is not to suggest that decoupling the conditions from the transformation make them easier to formulate but we do believe it makes it easier to correct erroneous conditions in the field. Also it is more convenient to add new transformations.

The second transformation we provide is the `MetricsTransformation`. The idea behind this is based on the work of McQuillan (2011) but the implementation provided here is our own. In this transformation, we maintain a representation of the system using McQuillan's *measurement* metamodel. The measurement metamodel contains a view of the measurable aspects of the system that are used by McQuillans *metrics* metamodel. The metrics metamodel contains definitions of a variety of metrics as OCL expressions. When these expressions are evaluated they provide a numeric value. This transformation also provides a 'script handle' called `$metrics`. This is significant because it means that measurement values can be reasoned about in pre and postconditions written in scripts. In other words, RefDstl allows the script developer to define refactorings that will only be applied in the case where the quality of the software undergoing refactoring is improved.

Condition Evaluator

From the RefDstl code examples previously shown, it is clear that there is a disparity between the syntax used to express pre and post conditions and the syntax used to describe refactoring operations. The disparity exists as a result of a language design choice to use a syntax for the conditions that would be familiar to programmers who have experience expressing pre and post conditions in other languages such as Spec#. However, this syntax is unsuitable to describe refactoring operations so we devised a declarative style reminiscent of SQL. It could be said that our language is really two languages in one and we held this in mind during the framework's design. For the sake of modularity, we placed the conditions evaluator into a separate component for this reason. The interpreter for evaluating conditions is called `ConditionInt`. Other than this class, the component contains one condition evaluator for each of the five levels of operator precedence in the RefDstl language. The `ConditionInt` delegates any condition to be evaluated to the interpreter class at the correct precedence.

Environment

`Environment` is responsible for maintaining state during script execution. It does this by providing a stack interface to clients. The stack is used much in the same way as in assembly programming to maintain local variables and for passing parameters to procedures. However, our implementation allows the client to explicitly create and destroy frames (which is done prior to a procedure call and immediately after). This prevents erroneous access of variables that are out of scope. Our environment does not need to provide a heap to store objects. Objects in RefDstl are just Java objects; hence they 'live' on the heap maintained by the JVM. When a reference to one of the objects is no longer on one of the stack frames maintained by `Environment` then that object can no longer be accessed so the object is garbage collected.

All variables that can be declared by RefDstl programmers have local scope. These can be accessed anywhere in the body of the procedure or in the conditions annotated on the procedure. Variables can also be declared as part of `forall` or `exists` constructs in conditions. These are only available in the body of the condition. The same applies to variables declared in the guard of `if` constructs. In loop constructs, variables can be declared in the header. These are available anywhere in the body of the loop construct including inside the bodies of any nested constructs. There are certain variables that are global. These are provided by 'script handles' (discussed later). For example, the `MoDiscoTransformation` provides a handle called `$project` that allows users to reason about the program that is being refactored. Although local and global variables are accessed and modified the same way, there is one major distinction in how they need to be treated in post conditions on procedures. In post conditions, it will be necessary to talk about the state of global variables prior to the procedure being executed so that conditions can be written to ensure that the model was refactored correctly. The environment supports features to enable this. It allows a deep copy to be made of a frame and of the objects/values in that frame. To access the old values, RefDstl programmers can use this syntax: `old( $someGlobalVar )` to gain access to the value of `$someGlobalVar` prior to the procedure being executed. Local variables cannot be reassigned in the RefDstl language so they have no 'old' state.

Script Handles

The `ScriptHandles` component contains extension points for the RefDstl language. Other than when the RefDstl framework 'starts up', and registers each of the handles with the `Environment` component, the other components are unaware of its existence. Each script handle simply becomes another global variable. We discuss how script handles are developed in the next section.

4.2 Enriching the RefDstl Language via Script Handles

Script handles are one of the extension points on offer to developers extending the basic functionality of the RefDstl framework. They can be included to either provide additional features in the language or as a substitute for 'macros' so that common expressions can be expressed succinctly. For example, Listing 4.5 shows how a script handle we included can be used to reduce the number of lines of code required to reason over all of the classes in the system being refactored. Without this script handle, we would have to use just the methods provided by the MoDisco representation of the system under refactoring, as shown in Listing 4.6.

```
requires forall $cls in $utils.allClasses() : ( . . . )
```

Listing 4.5: Writing a precondition that uses the utility method in the $utils handle for convenience.

```
requires forall $cu in $project.getCompilationUnits() : forall $cls in $cu.
   getTypes() : $instance.Of($cls, ClassDeclaration) -> ( . . . )
```

Listing 4.6: Pure RefDstl code for universally quantifying over all the classes in a system

In this section, we discuss how developers can integrate their own script handles into the framework. We then briefly describe four of the script handles that we developed. Note that script handles are accessed via variables that are treated as global by the environment.

4.2.1 Embedding a New Handle

Embedding a new script handle into RefDstl involves following these steps:

1. Create a Singleton Class — The steps involved for creating a singleton class are described by Metsker and Wake (2006). Roughly speaking it involves providing a default constructor with a private access modifier so that instances of the class can only be created from inside the class. The class also contains a reference to an object (having the same type as the class) which is marked static and private and is initialised at declaration. This ensures that only one object with that class's type can ever exist. The object can be accessed from outside the class by creating a static method (normally called `instance()`).

2. Implement the interface presented in Table 4.1 — The script handle must implement this interface so that the framework registers it correctly. The `getHandleName` method should return a `String` object. This will be the name of the variable that RefDstl script programmers use to access the script handle. `getHandleObject()` returns an instance of object. This should be the singleton object that is created as a part of enforcing the singleton design pattern. The class is free to contain any other methods. Methods that are declared public can be accessed in the script using the script handle.

Method Name	Purpose
getHandleName	This provides the name of the handle that can be accessed through the conditions. It is the responsibility of the person writing the script handle to make sure that the handle name is not already in use.
getHandleObject	This should return the instance of the handler class.

Table 4.1: The interface that must be implemented by a class to integrate it into the RefDstl framework as a script handle.

3. Register the handle in the framework — The script handle can be registered in the framework by modifying the constructor of the class `MachineStack` to include a line such as Listing 4.7. The `MachineStack` will then include a variable on every stack frame that points to the script handle's instance using its provided variable name. In the current implementation of RefDstl, there is no convenient way to determine what script handles are used in some script. In a future version, the interpreter could perform a static analysis of the script prior to execution to look for undeclared variables. Any undeclared variables could be suggested to be required script handles. However, this is not a guarantee as the script's author may have just misspelled a declared variable.

We discuss the script handles we implemented next.

```java
scriptHandles.add(Utils.instance());
```

Listing 4.7: Java code that is used to register the utils script handle object with the RefDstl framework

4.2.2 Supporting Debugging in RefDstl Scripts

There are no debugging tools currently available for RefDstl scripts nor is there any native support for I/O in the language. For this reason, we introduced a crude form of debugging by developing a script handle that allows tagged messages to be written to the console and can also be processed by the Log4J system[1].

The plug–in can be accessed by using the `$debug` handle. It offers a method named `print` that takes a message to display or a tag followed by a message. A tag should be any of the Log4J levels such as trace, debug, error, fatal, warn or info. Depending on the tag, the debug message will be output at the most appropriate Log4J level and to the configured logger.

4.2.3 Checking Object Types

While writing the standard library of primitive refactoring operations we found that it was useful to be able to test the type of objects. For this purpose, we provided the `instance` script handler. It contains a method called `of` which accepts two parameters. The first argument should be an object and the second should be a string that is a type name. The method will test if the object is an instance of that type name and return the result as a boolean.

4.2.4 Accessing the Program Model

The model of the system can be accessed using the `$project` handle. This allows for full access to the project under refactoring's MoDisco representation. Any methods provided by

[1]http://logging.apache.org/log4j/2.x/

the Java class implementation of MoDisco are available. For example, users can call the `getCompilationUnits` method to get the list of Java files in the project being refactored.

4.2.5 Accessing Old Variables

As discussed previously, global variables require two states to be maintained during the execution of a procedure in order to support postconditions: the state prior to execution and the post state. In a postcondition expression, the post value of the variable can be expressed by using the variable name — no special syntax is needed. However, the value of the variable prior to execution requires extra syntax. We provide this by using a script handle. The handle offers a single method called `valueOf` which takes a string object as a parameter. It reads the value for that variable from the frozen frames of the `MachineStack` and returns the value.

4.3 Custom Transformations into the RefDstl Framework

Users can integrate their own transformations into the framework by creating a singleton class that implements the `ITransformation` interface. The methods of this interface are documented in Table 4.2.

The user can create transformations for any type of representation that they choose but it is expected that all representations loaded during the execution of a script are just different views of the same model. However, there is no necessity for them to have been generated from some common core model.

We offer no guidance here regarding how to write the actual transformation code; this will depend very much on the representation the transformation is intended to refactor.

Method Name	**Meaning**
nop	In situations when a RefDstl programmer would like for a procedure, conditional or looping body to be empty they use a nop instruction. Transformations are not required to perform any action when a `nop` instruction is encountered.
save	A transformation is expected to have a reference to some model representation. The transformation should implement the save method to persist that model to a file or otherwise.
deleteClass	The script encountered a delete class instruction. The transformation should update the model representation by removing the class and any references to it.
deleteField	A delete field instruction was executed. The transformation should delete the field from its representation and any references to it.
createClass	A request to create a new class was encountered in the script. The model representation should create a new unreferenced class with the specified parameters.
renameClass	The script contained a rename class instruction. The transformation should modify the model by finding the class with the old name and update it with the new name.
createField	A new and unreferenced field should be created in a specified class.

createMethod	The transformation should create a new unreferenced method in the model representation.
deleteMethod	The transformation should search the model for a method with a specific signature and delete it as well as any associated references.
renameField	The field with a specific name should be found and its name updated to the name provided by the parameters.
changeMethodName	The model transformation should find a method with a particular name and signature and update its name.
changeFieldType	The type of an existing field should be changed by the transformation.
changeMethodType	The return type of an existing method should be changed in the representation held by the transformation.
changeMethodModifier	The access modifier of a method should be modified in the representation maintained by the transformation.
changeFieldModifier	An existing field should have its access modifier updated by the transformation.
addMethodArgument	An existing method should be modified so that a new argument appears in its signature.
deleteMethodArgument	An existing method should have its signature augmented so that a specified parameter is deleted.
reorderMethodArgument	The signature should be rewritten so that the arguments to a method are reordered.
changeSuperclass	An existing class should have its superclass changed from the current class to the specified one.
moveFieldToSuperclass	A field should be moved by the transformation from a subclass up the inheritance hierarchy into a superclass.
moveField	The transformation should relocate a field from the class that it is currently defined in to a new class.
moveFieldToSubclass	A copy of an existing field should be moved by the model transformation from the class it is currently located in to each of the subclasses.
moveMethod	A method should be moved from one class to another.
moveMethodToSuperclass	A method should be moved from a class to its superclass.
moveMethodToSubclass	A method should be moved from a class to its subclass.

Table 4.2: The methods that must be implemented by the transformation interface.

4.4 Disadvantage of Our Approach

The powerful extensibility of our framework is also its biggest disadvantage. RefDstl scripts that will execute on one person's installation might not work on another installation. We propose in the future to provide a mechanism to load script handles and transformations dynamically so that the dependencies of a script can be bundled with the script.

4.5 Summary

In this chapter, we presented the architecture of the RefDstl framework and identified the ‘key’ components. We emphasised how users may extend this framework using two approaches. The first involves creating script handles that enrich the scripting language with additional functionality, for example, interacting with the user via the console. The second involved incorporating additional model transformations, which are the primary reason the framework is capable of working on a variety of heterogeneous artefacts.

This chapter in conjunction with Chapter 3 concludes our discussion on the implementation of the RefDstl system[2]. In Chapter 5, we concentrate on evaluating the efficacy of the system as well as the accuracy of the model transformation used to refactor MoDisco models representing Java systems.

[2]By ‘system’, we mean the language and the framework.

Chapter 5

A Case Study on Automatically Refactoring 'God' Classes

Up to this point, we have provided the motivation for a DSTL to script refactorings and to define new composite refactorings. We presented the syntax of the RefDstl language and described how the system can be extended with custom model representations and transformations.

Now we focus our attention on evaluating the efficacy of our approach and our prototype implementation. We do this by performing a case study on a collection of 'God' classes selected from a corpus of Java software. At a low level, the study shows that: 1. RefDstl can be used to remove the 'God' class 'smell' using the extract class refactoring, and, 2. RefDstl can be used to measure coupling between Java classes. From a higher level, however, it is shown that RefDstl is suitable in practical environments to refactor 'real world' sized software and in academic environments as an extensible framework to evaluate research hypotheses.

5.1 Purpose of Experiment

In order to demonstrate the efficacy of RefDstl, we aim to show that it has the following qualities:

- Expressibility — We show that RefDstl is capable of expressing refactorings on a scale on par with what would be required by professionals and academics alike.
- Efficiency — Although our current implementation is only a prototype, it must be capable of performing real world refactorings in a reasonable amount of time. We follow the direction of Simon et al. (2001) who state that a refactoring engine should be able to perform the refactoring quicker than an experienced engineer performing it manually. Quantitative data is collected during the study and presented here that shows the time taken to perform the refactorings.

These objectives imply the following criteria should be considered when choosing the refactoring to be used in this study.

- The refactoring should resolve a 'naturally' occurring symptom of 'code rot'. This criterion helps to serve as evidence that RefDstl is useful in 'real world' situations.
- The refactoring should affect multiple different entities in the system, i.e. classes, methods and fields. It should also involve multiple refactoring primitives. This ensures that the range of refactoring instructions provided by RefDstl are broad enough to handle reasonably

complex refactorings. It also allows us to consider the time taken by RefDstl to perform a reasonably large refactoring.

With this criteria in mind, we examined the literature and discovered that as classes age they become less cohesive; they effectively become 'God' classes. The literature discusses approaches to address the issue. For example Simon et al. (2001) use distance metrics to present visualisations to software engineers as a guide of where to apply refactorings. However, we believe RefDstl provides better opportunities, which are the focus of this study. We use RefDstl to automatically remove the 'God' class 'smell' from a selection of classes and explore the impact on the CBO metric (Chidamber and Kemerer, 1994) as well as its predictor by Chaparro et al. (2014). We also examine the time it takes to perform the refactoring. We discuss how the study was conducted next.

5.2 Summary of Approach

We use the following sequential activities identified by Mens and Tourwe (2004) as the basis for our approach. These include:

1. Identifying refactoring opportunities — Mens and Tourwe (2004) assume that the refactoring is being conducted by a software engineer working on their own project. However, in this case study we also have the additional step of selecting systems to be the subjects of the experiment. Once identified, we then identify classes that exhibit the 'God' class 'smell' using an analysis tool that is developed in section 5.5.

2. Choosing a refactoring to apply — We intentionally look for 'God' classes in this study. Consequently, the refactoring to apply is predetermined to be the extract class refactoring.

3. Ensuring the program behaviour is preserved — It is the responsibility of the programmer writing the RefDstl refactoring script to ensure that that pre and post conditions for the refactoring are sufficient to preserve program behaviour.

4. Apply the refactoring — The RefDstl refactoring engine applies the extract class refactoring script to MoDisco models that represent the system under refactoring. These models contain the 'God' classes. We do not do this manually.

5. Assess the effect of refactoring on quality characteristics of software — We investigate the impact of extract class on the CBO metric. We examine if the *post* CBO measurement of the class under refactoring can be predicted.

6. Maintain consistency with other software artefacts — This does not form a part of our study. However, it should be noted for all other artefacts that are representable by a model that these can be manipulated by the RefDstl engine *during* refactoring.

We discuss the first of these activities next.

5.3 Identifying a Corpus for the Experiment

Ideally this study would be carried out on commercial software systems. However we note that more often than not commerical code is proprietary and industry is unwilling to share their code to maintain their competitive advantage. In experiments where commercial code has been used, the researchers are normally subjected to non–disclosure agreements that oftentimes require they do not name the product or further disseminate the code, for example in the study by Barker and Tempero (2007). This poses a problem for researchers where the subject of the experiments should be made available so that the experiment can be repeated and that fair comparisons can be made later with other studies. As an alternative we use a corpus of open source software. We also choose to restrict ourselves to using Java source code because we only have access to a MoDisco knowledge discoverer for transforming Java into MoDisco models.

We have identified the following potential sources to use in the experiment. Blackburn et al. (2006) proposed the DeCapo benchmark suite, which is described as 'set of open source, real world applications with non-trivial memory loads'. Although it is primarily a suite of previously calculated and verified benchmarks and measurements (such as object–oriented metrics) the software systems that the measurements were taken from can be downloaded using the Ant[1] script in the source–code distribution. However, we choose not to use this suite since it is an older corpus than the Qualitas Corpus, which we discuss next.

Tempero et al. (2010) provide the curated Qualitas Corpus as three distributions: The `r` distribution provides the most recent version of a large collection of software systems. The `e` distribution is the evolutionary version. It provides fewer systems but each of the systems is provided with multiple versions. The `f` distribution completes the corpus with systems and versions that appear in neither of the other two distributions. We select the `e` distribution for this experiment as it is suitably large and fits the requirements for our evolutionary study. The following section provides a summary of the contents of the corpus and describes its size. We note that there exists multiple releases of the corpus that come with later releases of software with substitutions made for systems that are no longer actively maintained. From this point on, we refer to version 20130901e when discussing the Qualitas Corpus.

The Qualitas Corpus has gained acceptance among other researchers such as the following. Fontana et al. (2012) perform a study on the affects of individual code 'smells' on different design metrics. Griffith et al. (2014) examine the relationship between technical debt estimation models and quality models. Al-Mutawa et al. (2014) examine whether all circular dependencies should be considered as 'undesirable'. De Roover et al. (2013) augment the Qualitas Corpus in the form of Quaatlas so that it is more suitable for studies concerning API usage. Arcelli et al. (2015) use the corpus to discover what refactoring tools are easiest and most useful for removing code 'smells'.

5.4 Content of the Qualitas Corpus

We briefly remark on the contents of the Qualitas Corpus in order to provide a sense of the size of this study. The corpus is disseminated as a split tape archive file, which is approximately 16.75GB in its unpacked state and 66.41GB in its decompressed installed state. It contains fifteen different systems from different domains. These are listed in Table 5.1 along with the number of versions of each system included. In total there are 579 versions of software and 3,320,594 Java classes (which are distributed across the various systems as shown by Figure 5.1). A variety of systems offers a degree of certainty that projects have disjoint sets of developers working on them. If the

[1]http://ant.apache.org

projects were developed by the same developers then the corpus would be unsuitable for certain types of studies such as those sensitive to developer habits. This includes the study presented here.

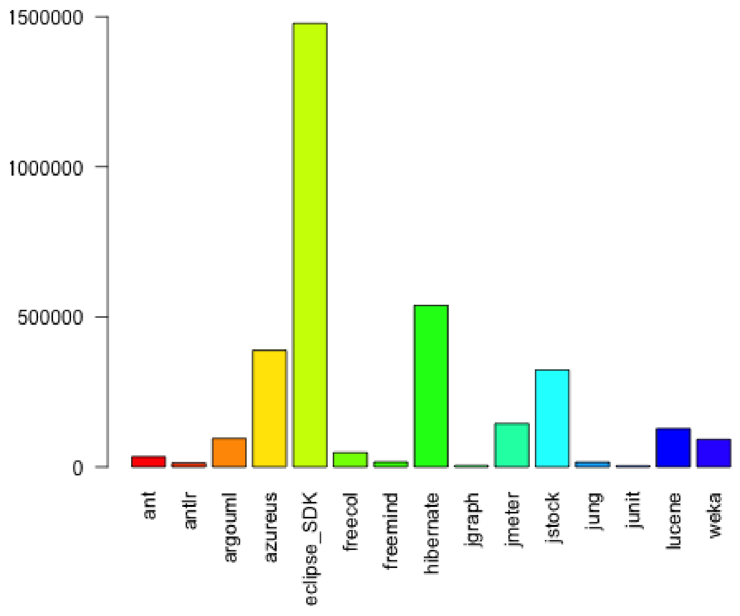

Figure 5.1: The number of classes per system.

The corpus contains the binaries and source code for each system as well as a summary of the corpus content and a summary of each project (i.e. metadata). Each project summary includes:

- A listing of the classes in the project and the location of each of those classes.
- A count of the number of lines of code in each source file and non–commented lines of code.
- An indicator determining whether each class is public or private and top level or nested.

Clearly not all of the classes provided with the corpus have the 'God' class 'smell' and some processing is required to identify the relevant classes. We develop an analysis tool for this purpose in the next section.

5.5 An Analysis Tool for the Qualitas Corpus

We develop a tool for analysing the content of the Qualitas Corpus. Its purpose is to perform two activities.

- It produces a relational model of the Qualitas Corpus metadata. This allows us to query the contents of the corpus easily and efficiently using SQL, rather than writing several ad hoc programs to process the files which is the default format for the corpus metadata. It also

System Name	No. of Versions Included
Ant	23
Antlr	22
ArgoUML	16
Azureues	63
Eclipse SDK	52
Freecol	32
Freemind	16
Hibernate	115
JGraph	39
JMeter	24
JStock	31
Jung	23
JUnit	24
Lucene	36
Weka	63

Table 5.1: Systems and number of versions provided in the Qualitas Corpus version 20130901e.

allows us to succinctly describe what subjects were used for the case study as an SQL query, rather than providing a long list of class and system version names.

- It performs an analysis of the classes in the system. The results of this analysis are persisted to the relational database and stored close to the metadata for later querying.

5.5.1 Persisting the Metadata

The Qualitas Corpus metadata is mapped using a text processing program written in Java to the simplified relational model shown in the entity–relationship diagram presented in Figure 5.2. This is implemented using the PostgreSQL[2] database system.

5.5.2 Analysing the Classes

Analysing software source code involves one of the following analysis approaches:

- Static — This approach involves analysing the code without executing it and is suitable in situations where either the compiled binary or just the source code are available.

- Dynamic — Dynamic analysis involves executing the program. Some binary analyses can be conducted without needing access to the source code, for example, analysing the time taken by a program to finish executing. However, other more involved analyses require that the code is instrumented before being executed. For example, suppose we want to generate a list of the methods that are called during the execution of a program with unknown inputs. Dynamic analyses can be slower because of the time that it takes to execute the program undergoing analysis.

We chose to incorporate static analysis into the tool. At the core of the analysis tool is an iterator that goes through each of the classes provided in the corpus. For efficiency reasons, it

[2]http://www.postgresql.org

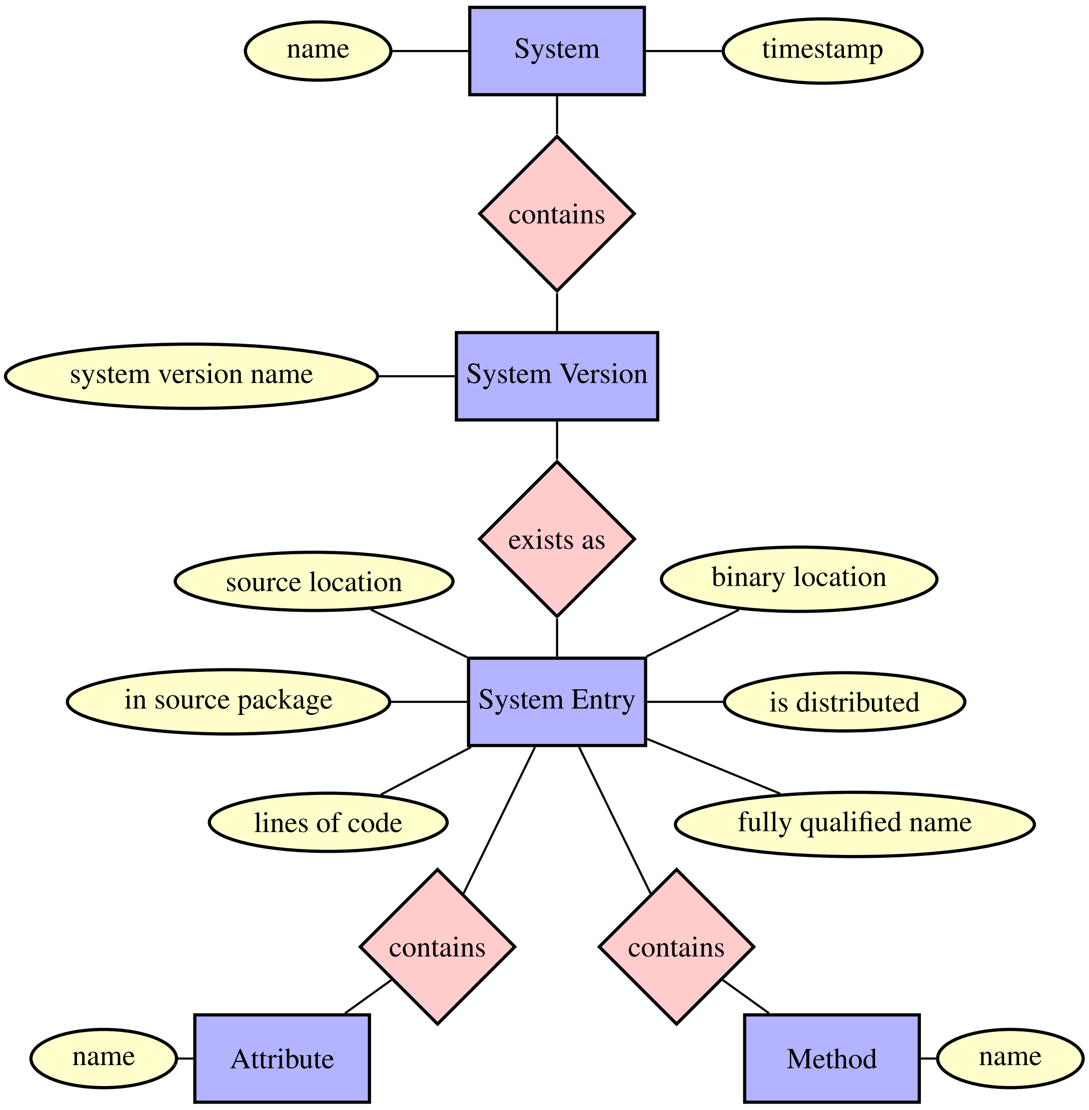

Figure 5.2: An entity relationship diagram illustrating the main entities and attributes persisted by the static analysis tool. Other entities, attributes and keys have been omitted for simplicity.

is designed to iterate over the classes stored in a Jar file sequentially. This has been done for efficiency because it means that a Java archive file only needs to be loaded and searched for Java class files once. While iterating over the classes, the actual analysis occurs. This requires using a library to analyse the classes. Two libraries were considered for this purpose: ASM (Bruneton et al., 2002) and BCEL [3].

ASM works on the level of source code. It follows the visitor design pattern so developers define actions to occur when nodes of a particular type are encountered. This is similar to the SAX streaming API for parsing XML. However, it also offers a less memory efficient but more user friendly DOM style interface that allows the developer to 'pull' what they want from a tree structure of the program.

We opted however to use BCEL. It works at the lower byte–code level (although the byte–code does not need to be executed). BCEL of course requires that the program being analysed can be compiled. However, as mentioned earlier, the corpus provides the system binaries so this was not a problem. BCEL has an advantage over ASM in that some analyses are simpler. For example, with BCEL, we can search for patterns of bytecode so we can easily find all field accesses or mutations in a class.

Our static analysis computes class level metric measurements (TCC, WMC, ATFD) and saves the results to the database.

One might wonder why we developed a separate analysis tool for calculating these metrics rather than employing the RefDstl system. We could have used RefDstl for this purpose. However, initial experimentation showed that RefDstl would have been too slow to calculate the metrics for the three million classes in the corpus.

5.6 Selecting Refactoring Candidates

Our goal is to select the 'God' classes from the Qualitas corpus. However, we will choose only classes that appear in every version of a system distributed with the corpus and are 'God' classes throughout. This provides a degree of certainty that what we select are actually 'God' classes and not 'borderline' that could be reclassified due to a minor change occurring in some version.

To assess which of classes are 'God' classes, we use the criteria described by Lanza et al. (2005). They define a strategy for finding instances of the 'God' class 'disharmony' using these requirements:

- access to foreign data (ATFD) (Lanza et al., 2005) metric is greater than a few, and,
- weighted method per class (WMC) (Chidamber and Kemerer, 1994) metric is very high, and,
- tight class cohesion (TCC) (Bieman and Kang, 1995) metric is less than a third.

For the values 'greater than a few', we use the value 1, for very high we use 25%. This is in line with the values given in Marinescu (2004). The query to identify 'God' classes that appear in every version of a system is presented in Figure 5.3.

[3]http://commons.apache.org/proper/commons-bcel/

```sql
CREATE VIEW god_classes_all_versions AS
SELECT T.*, system_entry.*
FROM ( SELECT se.system_name, se.fully_qualified_name
       FROM system_entry se
       WHERE se.tcc < 0.33
          AND se.atfd > 1
          AND se.wmc > 25
          AND se.system_name <> 'eclipse'
       GROUP BY se.system_name, se.fully_qualified_name
       HAVING count(se.fully_qualified_name) = ( SELECT count(sv.*)
                                                 FROM system_version sv
                                                 WHERE sv.system_name
                                                       = se.system_name
                                                 GROUP BY sv.system_name )
) T
JOIN system_entry USING (system_name, fully_qualified_name);
```

Figure 5.3: SQL query to discover 'God' classes in all versions.

```sql
SELECT *
FROM god_classes_all_versions
WHERE system_version_version IN ( SELECT *
                                  FROM max_version )
   AND system_name IN ( 'ant', 'argouml', 'freemind', 'jgraph', 'jmeter',
                        'jstock', 'jung', 'junit', 'lucene', 'weka' )
   AND system_version_version NOT IN ( 'argouml-0.16.1', 'argouml-0.18.1',
                                       'freemind-0.6.7', 'weka-3.4',
                                       'weka-3.4.12' )
   AND level = 'Top Level Public'

```

Figure 5.4: SQL query to discover classes for the study.

This dataset had to be later narrowed down to the dataset given by the query in Figure 5.4. This was due to either the systems being too large to analyse (Eclipse) or being unable to generate models for those projects (which is discussed later).

5.6.1 Creating the MoDisco Models

A two stage process was involved to produce the MoDisco models for the projects in the Qualitas corpus:

1. We wrote a Java program that converted each system version in the corpus into a valid Eclipse project with the correct classpath settings. The classpath settings were created using the data pulled from the analysis tool described earlier.

2. With each system version in the form of an Eclipse project, we could produce MoDisco models. The MoDisco plugin for Eclipse comes with knowledge discoverers to do this. However, out of the box, these are designed to be interacted with manually. We developed a plugin for Eclipse that performed the knowledge discovery and saved the MoDisco models without human interaction. In total, it takes longer than six hours for the plugin to convert every project (ignoring the Eclipse SDK). The time spent developing the plugin paid for itself considering we had to generate the models three times before we were left with a

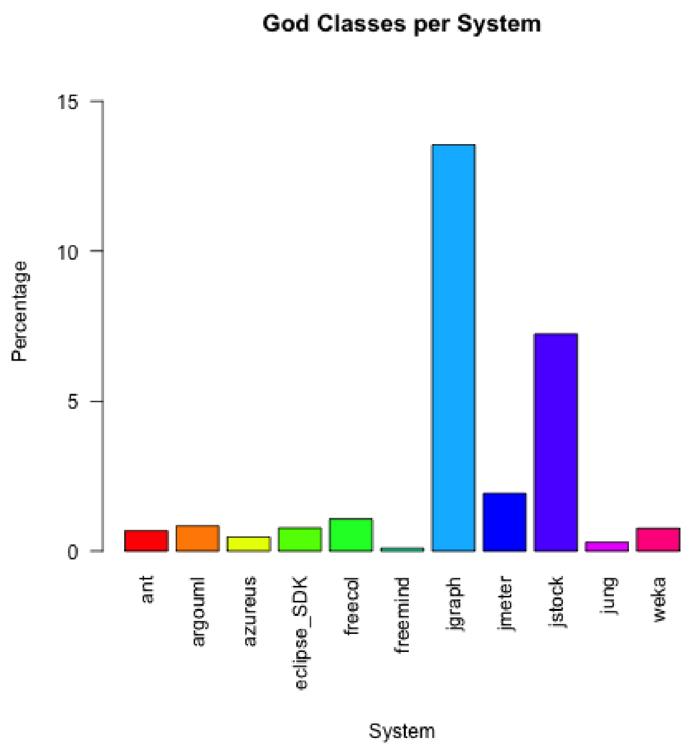

Figure 5.5: The percentage of 'God' classes for all versions of each of the software systems in the corpus.

reasonable number of models. We did not produce the models correctly the first two times because the classpath settings for the projects were initially incorrect. On our third attempt we managed to produce 458 valid models out of 579 systems. The large majority of the systems we could not produce models for were the hibernate system. We therefore excluded all hibernate systems from our study along with certain versions of the other systems. The dataset that was actually used for the study is given by the query in Figure 5.4.

In the two following sections, we discuss our protocol for deciding how to split an incohesive class as well as the RefDstl extract class script for performing the extraction.

5.7 An Extract Class Refactoring

Applying an extract class refactoring to an incohesive class involves moving elements of the incohesive class to a newly created class. This is accomplished using a series of move field and method refactorings as well as create class refactorings. However, deciding what fields and methods to move automatically involves using algorithms such as those used in social network analysis like the Girvan–Newman algorithm (Cassell et al., 2009; Girvan and Newman, 2002) or from the area of machine learning using algorithms such as *k*–means. The algorithm we used is as follows:

1. For every method in the class that is being automatically refactored, we create a cluster to contain that method.

2. We iteratively check each cluster against the other clusters to determine if the clusters should be merged. We say that two clusters should be merged if greater than half of the methods in

Method Name	Parameters	Description
cluster	Class name	This method will cluster the class with the fully qualified name specified in the input paramaters. The method always returns true so it can be used safely within a pre or post condition expression.
fieldToClass	A field.	Returns the name of the class where the field should be moved.
methodToClass	A method.	Provides the name of the class to which the method should be moved.

Table 5.2: API for interacting with the clustering plugin.

the two clusters are similar. Two methods are considered similar if at least half of their field accesses are to the same fields.

3. After the clusters in the previous step converge, we then turn to allocating the fields to the clusters. Each field is assigned to the cluster that uses that field the most.

We implemented this algorithm in a script handle for the RefDstl framework since the RefDstl language has no way of expressing the algorithm otherwise. The API for using the handle is shown in Table 5.2.

A RefDstl script was created automatically for each of the classes to be refactored and identified in our dataset. An example of one of the scripts is shown in Figure 5.6. It shows the `main` procedure calling another to perform the extract class refactoring. This delegation is necessary since `main` does not support pre or post conditions.

The precondition is used to arrange the class elements into cohesive units: this is done using the call to the `clusterClass` method.

The first postcondition is used to calculate the CBO metric for the original class post refactoring. The formula for calculating the CBO value for a class is given in Equation 5.1. In words, it is a count of the number of classes a class c is coupled with. The last postcondition calculates the predicted value for CBO. This is done using the CBO predictor function by Chaparro et al. (2014), which we modified to take into account multiple extract class refactorings being performed at once. The formula is shown in Equation 5.2. It says that the CBO value for a class that has undergone an extract class refactoring should be equal to its original value plus n which is a count of the number of classes that were extracted less the number d which is a count of the classes coupled to by the extracted methods but not used by the original class. In the original formula proposed by Chaparro et al. (2014), n has the constant value of one.

$$CBO(c) = |c_{coupled}| \tag{5.1}$$

$$CBO_p(c_s) = CBO_b(c_s) + n - d \tag{5.2}$$

The results of applying the clustering algorithm to the classes that were selected for refactoring are discussed next, which includes comparing the actual CBO measurements to their predicted counterparts.

```
in 'model_java2kdm.xmi'
out 'model_out.xmi'

main()
{
 call measureAndSubvert('org.apache.tools.ant.DirectoryScanner')
}

proc measureAndSubvert($originalGodClass)
 requires $clusterer.clusterClass($utils.findClass($originalGodClass))
 ensures $debug.printToFile('/Users/keith/Desktop/metrics.txt', 'ant', 'ant
     -1.8.4', 'org.apache.tools.ant.DirectoryScanner', $ripe.cbo($utils.
     findClass('org.apache.tools.ant.DirectoryScanner')))
 ensures $debug.printToFile('/Users/keith/Desktop/metrics_pred.txt', 'org.
     apache.tools.ant.DirectoryScanner', 'ant', 'ant-1.8.4', $ripe.cboPredicted(
     $utils.oldClass('org.apache.tools.ant.DirectoryScanner'), $utils.findClass
     ('org.apache.tools.ant.DirectoryScanner')))
{
 for($extractedClassName in $clusterer.classes()) {
  if(~$extractedClassName.equals($originalGodClass) {
   call createClass($extractedClassName)
  }
 }

 for ($method in $utils.findClass($originalGodClass).getMethods()) {
  if(~$originalGodClass.equals($clusterer.methodToClass($method)) {
   call makeMethodPublic($method.getName(), $method.getParamTypes(),
     $originalGodClass)
   call moveMethod($method.getName(), $method.getParamTypes(),
     $originalGodClass, $clusterer.methodToClass($method))
  }
 }

 for ($field in $utils.findClass($originalGodClass).getFields()) {
  if(~$originalGodClass.equals($clusterer.fieldToClass($field)) {
   call makeFieldPublic($field.getName(), $originalGodClass)
   call moveField($field.getName(), $originalGodClass, $clusterer.fieldToClass(
     $field))
  }
 }

 for($extractedClassName in $clusterer.classes()) {
  if(~$extractedClassName.equals($originalGodClass) {
   call createField($extractedClassName.toLowerCase(), 'private',
     $originalGodClass, $extractedClassName)
  }
 }
}
```

Figure 5.6: RefDstl sourcecode used to perform the extract class refactoring on an identified 'God' class.

5.8 Results

We discuss the results of the case study from two angles: the correlation between the actual CBO of the measured class post refactoring and the correlation between the the time taken to perform the refactoring and the size of the system being refactored.

5.8.1 Predicting CBO

Figure 5.7 summarises the results of the case study. The x–axis represents the predicted value of the CBO metric while the actual value is shown along the y–axis. The diagonal blue line shows a linear model produced from the data. Ideally, there would exist a perfect correlation between the actual measurement and the predicted measurement. This would result in a slope of 1, i.e. when the x value increases by a single unit then the y value increases by an equal amount. The slope presented here is 0.6843. There are two reasons why we might have fallen short of 1. The first reason are potential flaws in our implementation. For example, the refactoring transformations that were implemented in the interpreter could have coding defects due to human error that is typical when developing any software system. Secondly, the predictor function is inaccurate. We note that in their evaluation, Chaparro et al. (2014) tested this particular CBO predictor function by having two postgraduate students perform between five and ten refactorings on two software systems. When the predicted and actual CBO measurements were compared for the two systems just 63% of the measurements were completely accurate.

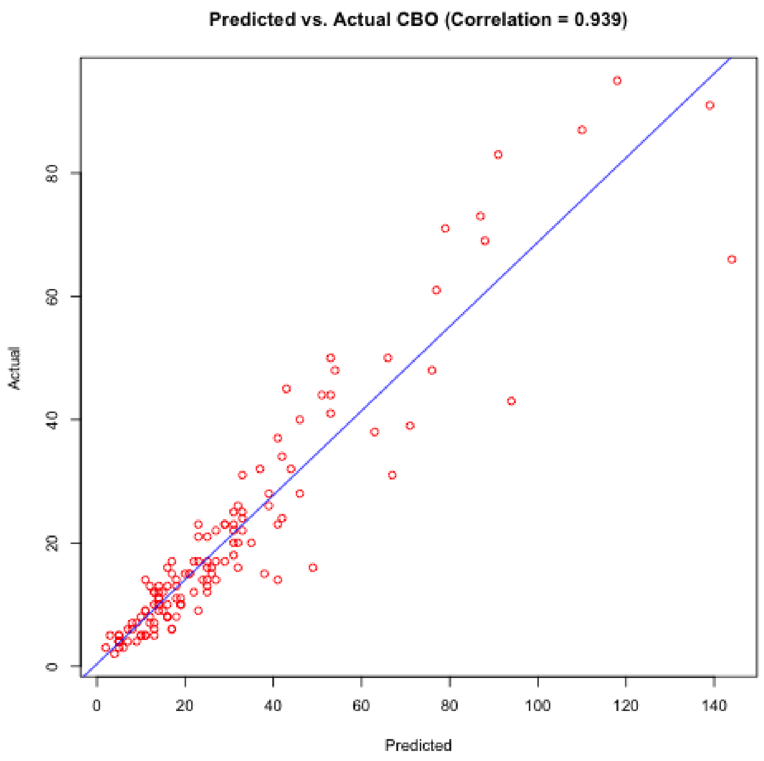

Figure 5.7: Pearson's correlation between the predicted CBO metric and the actual value .

We can further measure how close the predicted value correlates with the actual value using

Pearson's correlation coefficient, which is given by the `cor` function in the `R` language[4] for statistical computing. In our results, the coefficient is 0.939. A perfect correlation would yield a value of 1 but nonetheless this value shows high correlation.

We note that there are some clear outliers in this data. The most apparent is due to the class 'org.yccheok.jstock.gui.MainFrame' in 'jstock-1.0.7c'. Its actual value after refactoring is 66 but its predicted value is 144. Further investigation, which involved reading the source code of that class, would suggest that we are not handling the case correctly where class attributes are accessed using the `this` Java keyword.

We remark on the distribution of the CBO measurements as shown by the three 'box–and–whisker' plots in Figure 5.8. In each plot, the 'whiskers' at the top and bottom of the plot represent the range of values (discarding the outliers shown as red coloured points in the figure). The thick black band inside each of the coloured boxes represents the median for that dataset while the beginning and end of each box represents the first and third quartiles for each dataset respectively. A perfectly distributed dataset would see the median located half way between the 'whiskers' while the first and third quartiles would be at the 25% and 75% marks. What's interesting, however, is how the plots compare to each other. We can see that the predicted values are generally higher than the original values. The outliers have higher values and the first quartile is at a higher value. We would expect this to be the case because for each class that undergoes an extract class refactoring it gains a newly created field that references the extracted class. In contrast, the actual values tend to be lower than the predicted values. The actual outliers have lower values and the quartiles are also lower. We have no rationale to explain this. However, it suggests the presence of a 'bug' in our implementation: most likely we are not considering some class usages. These would increase the value of d in Equation 5.2 and hence reduce the predicted values.

We do not compare our results with those from the study by Chaparro et al. (2014). Doing this would be meaningless for the following reasons:

- We did not use the same dataset. Those authors restrict themselves to ArgoUML[5] and aTunes[6] (although they do not specify what version of each). On the other hand we use classes from a wider range of projects, including ArgoUML.
- We did not use the same measurement tool.
- We measured different representations of the systems (for example, the transformation for turning Java code into MoDisco models may have some bugs.)

5.8.2 Time Taken

As shown in Figure 5.9, there also exists a strong correlation (0.981) between the time taken to perform the refactoring and the size of the model. This supports the idea that most of the time spent by the refactoring engine goes toward loading the model to be refactored and writing it back to disk after refactoring. There is some small variation in the time taken to refactor models of different sizes. However, the greatest variation is around seven seconds. We consider this to be

[4]http://www.r-project.org
[5]http://argouml.tigris.org
[6]http://www.atunes.org

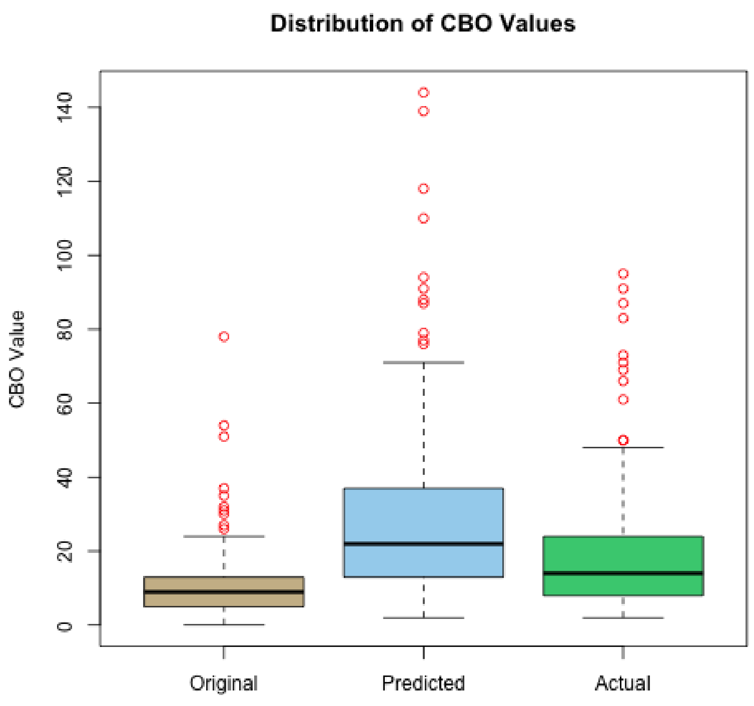

Figure 5.8: The distribution of the original, predicted post refactoring and actual post refactoring CBO measurements for the software systems used in the study.

negligible and bear in mind that these timings reflect 'wall time' and not 'CPU time' so slight variations might be caused by the operating system switching to background processes during the refactoring etc.

Overall the time taken to perform the refactorings was sixteen minutes. We do not believe that a developer could perform these 134 refactorings in this amount of time: deciding how to refactor just one 'God' class would probably take a developer as long.

We note that in this study we used Java code to calculate the CBO metric and its predicted counterpart. We had tried to use the metrics transformation also provided with RefDstl but we found that this was too slow (approximately six minutes to do one refactoring). The slowdown is due to having to transform every class to the metrics metamodel rather than just the needed classes. Future versions of RefDstl will address this weakness.

5.9 Discussion

We believe our case study shows that RefDstl is a suitable system for targeting the 'four Is' that were identified in Chapter 1:

1. The correlation between CBO and the predicted value for CBO suggests that our implementation has some degree of accuracy. If there were no correlation then this would suggest that either the prediction model is wrong or the implementation is flawed. However, given the results it would require a coincidence to get this correlation. A refined implementation should yield better results still.

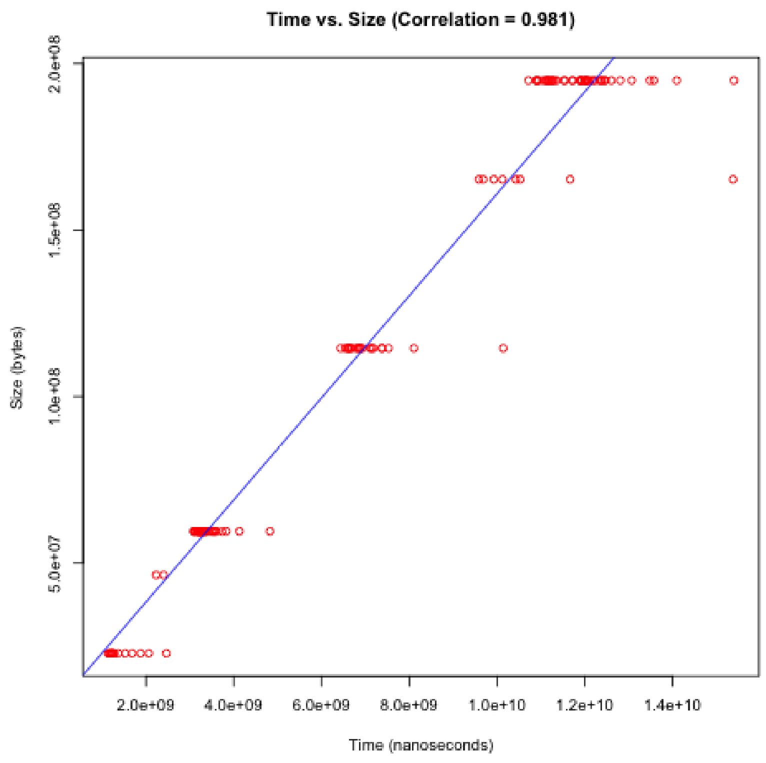

Figure 5.9: Pearson's correlation between the size of the model being refactored and the time taken to refactor the model in nanoseconds.

2. We used RefDstl to define a new composite refactoring that can be reused and shared in a library. This demonstrates that our approach affords flexibility to refactoring unlike many mainstream refactoring engines.

3. Our framework is extensible since it considered CBO metrics during the refactoring and can be adapted to consider other metrics or representations.

4. RefDstl had the foresight to know which fields and methods to move. This is dissimilar to other 'blind' refactoring engines that merely follow the (mis–)guidance of developers who could make incorrect decisions.

5.10 Summary

In this chapter, we have presented a case study of refactoring 134 'God' classes from the Qualitas Corpus. We refactored the classes automatically using an algorithm that we embedded into the RefDstl system. We evaluated our approach by comparing the CBO measurements post refactoring with predicted values. The high correspondence between the values provided some guarantee (although not absolute) about the accuracy of our refactoring system and the time measurements show the feasibility of our approach when considering time.

In the final chapter that follows we conclude this dissertation with a reflection of what has been achieved and how. We also remark on how this work might be expanded in the future.

Chapter 6

Conclusion

In this chapter, we highlight the contributions that have been made in this dissertation and we discuss topics that merit exploration in future work.

6.1 Summary of Contributions

Refactoring is the methodical approach to restructuring a software system such that its observable behaviour remains unchanged (Opdyke, 1992). In Chapter 1 and Chapter 2, we recognised the importance of refactoring engines in the refactoring process. However, we highlighted that these are susceptible to four recurrent weaknesses that we named the 'four IS'.

I1 referred to inaccuracy, which is when the conditions required for a refactoring to be applied are defined too strongly or weakly by the refactoring engine. In Chapter 2, we saw that attempts have been made to *formally* demonstrate that refactorings preserve behaviour for OO *specification* languages and for a small set of Java refactorings. However, the approaches taken by those researchers can only demonstrate that conditions are not weak. They cannot demonstrate that conditions are overly strong. We proposed that, until this issue is resolved rigorously, that it would be better if refactoring engines placed their conditions *away* from the transformation code so that conditions could be altered in 'the field'. For this reason, we developed the RefDstl language in Chapter 3. It can be used to specify new composite refactorings (using a DBC style) from provided primitive operations. The language also addresses I2 because the defined refactorings can be saved to scripts for later application.

With regard to I3, we implemented a framework for interpreting RefDstl scripts. It has been carefully designed so that additional transformations can be easily integrated. Each additional transformation is capable of refactoring a different type of artefact (although all artefacts should be different representations of the same model) and we provided two transformations for demonstration purposes: one to actually refactor models of Java systems and one for transforming the system under refactoring into a measurement metamodel developed by McQuillan (2011). By permitting additional transformations and allowing extension points called 'script handles', we allow RefDstl programmers to reason about different aspects or views of the system under refactoring in their refactoring pre and post conditions. This means that the RefDstl framework is not improvident like traditional refactoring engines (I4). It allows users to reason about the quality of the software during refactoring (which we demonstrated at the end of Chapter 3).

We evaluated the efficacy and, to a degree, the accuracy of our prototype implementation in Chapter 5. We showed how RefDstl could be used to automatically improve the structure of

'God' classes using a script handle that calculates the CBO value for the class under refactoring. The accuracy of our transformations was shown by checking the correlation between the actual CBO values post refactoring and the predicted values using a predictor function given by Chaparro et al. (2014). We explained that this was not a 'water tight' approach but a sensible heuristic. The predictor function could be incorrect and thus we would have achieved a low correlation anyway. However, it would require coincidence for either the predictor function or our transformations to be incorrect and to still achieve the high correlation that we did (0.939 using Pearson's correlation coefficient).

Also, as a starting point for those eager to use RefDstl, we have contributed a standard library of primitive refactoring operations in Appendix B. It was used in the examples throughout this dissertation.

Finally, we also want to highlight that our approach to developing the RefDstl language equips us with a reusable metamodel of the primitive refactorings described by Opdyke (1992). This can be used by researchers in areas beyond what has been described in this dissertation.

Now, having summarised our model driven approach for refactoring heterogeneous software artefacts, we briefly remark on avenues available for future work.

6.2 Future Work

In future work, we will take a different approach to tackle the problem of showing that refactoring preserves a program's behaviour. For Java programs equipped with JML specifications, we will integrate a transformation for JML annotations into the RefDstl framework. This will mean that when JML annotated programs are refactored that their specifications will also be refactored in tandem. By using theorem provers, we could then show that the specification of a program before refactoring is equivalent to the specification after refactoring.

For programs that are not annotated with specifications, we will offer an approach that allows users to reason about the structural properties of the software. We will develop the MoDisco metamodel in the Gallina language used by the Coq theorem prover[1] as a collection of structures[2]. We will integrate a transformation into the RefDstl framework that converts the system being refactored into instances of those structures (effectively a MoDisco to Gallina transformation). This would allow users to reason about the structure of their systems; for example, they could prove that all references to a renamed entity are preserved after refactoring.

Finally, we will enrich the interpreter with the capability to calculate the weakest precondition for any RefDstl procedure. This can be done using Dijkstra's Weakest Precondition Calculus but it would require that loop invariants are added to the RefDstl language.

[1]https://coq.inria.fr

[2]Similar to structs in 'C' or classes in Java.

Acronyms

ASCII	American Standard Code for Information Interchange.
AST	Abstract Syntax Tree.
ATFD	Access to Foreign Data.
ATL	Atlas Transformation Language.
BNF	Backus Naur Form.
CASE	Computer Aided Software Engineering.
CBO	Coupling Between Objects.
DBC	Design by Contract.
DSL	Domain Specific Language.
DSTL	Domain Specific Transformation Language.
EMF	Eclipse Modelling Framework.
IDE	Integrated Development Environment.
JML	Java Modelling Language.
LCC	Loose Class Cohesion.
LCOM	Lack of Cohesion of Methods.
MDE	Model Driven Engineering.
MOF	Meta Object Facility.
MPS	Meta Programming System.
OMG	Object Management Group.
OO	Object–Oriented.

QVTO Query View Transformation Operational.

QVTR Query View Transformation Relational.

SQL Structured Query Language.

TCC Tight Class Cohesion.

TIKZ Tikz ist kein Zeichenprogramm.

UML Unified Modelling Language.

WMC Weighted Methods Per Class.

XML Extensible Markup Language.

XSD XML Schema Definition.

Appendix A

RefDstl Language Grammar for Xtext

```
grammar ie.pop.refdsl.RefDsl

hidden(WS, ML_COMMENT, SL_COMMENT)

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

generate refDsl "http://www.pop.ie/refdsl/RefDsl"

RefactoringScript:
  requires += Import*
  ('in' project = STRING)?
  ('out' output = STRING)?
  (main = Main)?
  refactorings += Procedure*;

Import:
  'imports' file = STRING;

/* Procedures. */
Main:
  'main' '(' ')'
  '{'
     instructions += (RefactoringInstruction | ConditionalStatement |
   ProcedureCall | LoopingStatement)+
  '}';

Procedure:
  'proc' name=ID_PART '(' (arguments=ParameterList)? ')'
   ('requires' precondtions+=Condition1)*
   ('ensures' postconditions+=Condition1)*
  '{'
     instructions += (RefactoringInstruction | ConditionalStatement |
   ProcedureCall | LoopingStatement)+
  '}';

ProcedureCall:
  'call' name=ID_PART '(' args = ArgumentList? ')';

ConditionalStatement:
  'if' '(' condition=Condition1 ')' '{'
  instructions+=(RefactoringInstruction | ConditionalStatement | ProcedureCall
    | LoopingStatement)+
  '}';

LoopingStatement:
  'for' '(' id=VID 'in' (collection=ObjectExpression) ')' '{'
  instructions += (RefactoringInstruction | ConditionalStatement |
    ProcedureCall | LoopingStatement)+ '}';

RefactoringInstruction: {RefactoringInstruction}
  action=(CreateClass
  | CreateField
  | CreateMethod
  | DeleteClass
  | DeleteField
  | DeleteMethod
  | ChangeClassName
  | ChangeFieldName
  | ChangeMethodName
  | ChangeFieldType
  | ChangeMethodType
  | ChangeMethodModifier
  | ChangeFieldModifier
  | AddMethodArgument
  | DeleteMethodArgument
  | ReorderMethodArguments
  | ChangeSuperclass
  | MoveFieldToSuperclass
  | MoveFieldToSubclass
  | MoveMethodToSuperclass
  | MoveMethodToSubclass
  | MoveField
  | MoveMethod
  | NoOp);

/* Create syntax. */
CreateClass:
  'create' (accessor=Accessor)? 'class' name= (ID_PART | FULL_ID | VID) ('with'
     'superclass' superName=(ID_PART | FULL_ID | VID))?;

CreateField:
  'create' (accessor=Accessor)? (static='static')? 'field' name=(ID_PART | VID)
     'in' 'class' unit=(ID_PART | FULL_ID | VID) 'of' 'type' type=(FULL_ID |
    ID_PART | VID);

CreateMethod:
  'create' (accessor=Accessor)? (static='static')? 'method' name=(ID_PART | VID
    ) 'in' 'class' unit=(ID_PART | FULL_ID | VID) ('taking' '[' args=
    JavaParameterList ']')? ('returning' return=(FULL_ID | ID_PART | VID))?;

/* Delete syntax. */
```

```
DeleteClass:
  'delete' 'class' name=(FULL_ID | ID_PART | VID);

DeleteField:
  'delete' 'field' name=(ID_PART | VID) 'in' 'class' unit=(FULL_ID | VID |
    ID_PART);

DeleteMethod:
  'delete' 'method' name=(ID_PART | VID) 'in' 'class' unit=(FULL_ID | VID |
    ID_PART);

/* Change syntax. */
ChangeClassName:
  'rename' 'class' oldName=(FULL_ID | VID | ID_PART) 'to' newName=(FULL_ID |
    VID | ID_PART);

ChangeFieldName:
  'rename' 'field' oldName=(ID_PART | VID) 'to' newName=(ID_PART | VID) 'in' '
    class' unit=(FULL_ID | VID | ID_PART);

ChangeMethodName:
  'rename' 'method' oldName=(ID_PART | VID) 'to' newName=(ID_PART | VID) 'in' '
    class' unit=(FULL_ID | ID_PART | VID);

ChangeFieldType:
  'change' 'type' 'of' 'field' name=(FULL_ID | ID_PART | VID) 'in' 'class' unit
    =(FULL_ID | ID_PART | VID) 'to' newType=(FULL_ID | VID | ID_PART);

ChangeMethodType:
  'change' 'type' 'of' 'method' name=(ID_PART | VID) 'in' 'class' unit=(FULL_ID
     | ID_PART | VID) 'to' newType=(FULL_ID | VID | ID_PART);

ChangeMethodModifier:
  'make' 'method' name=(ID_PART | VID) ('with' 'args' '[' args=
    JavaParameterList ']')? 'in' 'class' unit=(FULL_ID | VID | ID_PART)
  newModifier=Accessor;

ChangeFieldModifier:
  'make' 'field' name=(ID_PART | VID) 'in' 'class' unit=(FULL_ID | ID_PART |
    VID) newModifier=Accessor;

AddMethodArgument:
  'add' 'argument' arg=JavaParameter 'to' 'method' name=(ID_PART | VID) ('with'
     'args' '[' args=JavaParameterList ']')?
  'in' 'class' unit=(FULL_ID | VID | ID_PART);

DeleteMethodArgument:
  'delete' 'argument' arg=(ID_PART | VID) 'from' 'method' name=(ID_PART | VID)
    ('with' 'args' '[' args=JavaParameterList ']')?
  'in' 'class' unit=(FULL_ID | VID | ID_PART);

ReorderMethodArguments:
  'place' 'argument' arg=(ID_PART | VID) 'in' 'method' name=(ID_PART | VID) 'with' 'args' '[' args=JavaParameterList ']' 'before' successor=(ID_PART | VID) 'in' 'class' unit=(FULL_ID | VID | ID_PART);

ChangeSuperclass:
  'class' unit=(FULL_ID | ID_PART | VID) 'extends' newSuperclass=(FULL_ID | VID | ID_PART);

/* Move syntax. */
MoveFieldToSuperclass:
  'move' 'field' name=(ID_PART | VID) 'from' unit=(FULL_ID | VID | ID_PART) 'up';

MoveFieldToSubclass:
  'move' 'field' name=(ID_PART | VID) 'from' unit=(FULL_ID | VID | ID_PART) 'down';

MoveMethodToSuperclass:
  'move' 'method' name=(ID_PART | VID) ('with' 'args' '[' args=JavaParameterList ']')? 'from' unit=(FULL_ID | VID | ID_PART) 'up';

MoveMethodToSubclass:
  'move' 'method' name=(ID_PART | VID) ('with' 'args' '[' args=JavaParameterList ']')? 'from' unit=(FULL_ID | VID | ID_PART) 'down';

MoveField:
  'move' 'field' name=(ID_PART | VID) 'from' unit=(FULL_ID | VID | ID_PART) 'to' new_unit=(FULL_ID | VID | ID_PART)
;

MoveMethod:
  'move' 'method' name=(ID_PART | VID) 'from' unit=(FULL_ID | VID | ID_PART) 'to' new_unit=(FULL_ID | VID | ID_PART)
;

NoOp:
  name = 'nop'
;

/* Helpers. */
Accessor:
  value=('public' | 'private' | 'protected' | 'default');

/*
 * Contracts.
 */
Condition1:
  left=Condition2 operator='->' right=Condition1
  | left=Condition2 operator='==' right=Condition1
  | left=Condition2 operator='~=' right=Condition1
  | left=Condition2 operator='>' right=Condition1
  | left=Condition2 operator='<' right=Condition1
  | left=Condition2 operator='>=' right=Condition1
  | left=Condition2 operator='<=' right=Condition1
  | higher=Condition2;

Condition2:
    operator='forall' id=VID 'in' collection=ObjectExpression ':' left=
    Condition1
  | operator='exists' id=VID 'in' collection=ObjectExpression ':' left=
    Condition1
  | higher=Condition3;

Condition3:
  left=Condition4 operator='/\\' right=Condition3
  | left=Condition4 operator='\\/' right=Condition3
  | higher=Condition4;

Condition4:
  operator='~' left=Condition4
  | higher=Condition5;

Condition5:
  '(' bracketed=Condition1 ')'
  | arg=ObjectExpression
  | constant=LogicalConstant
  | scalar=STRING;

/*
 * Parameters.
 */

JavaParameter:
  type = (FULL_ID | ID_PART | VID) name = (FULL_ID | ID_PART | VID)
;

JavaParameterList:
  parameters += JavaParameter (',' parameters += JavaParameter)*
;

Parameter:
  name = VID
;

ParameterList:
  parameters += Parameter (',' parameters += Parameter)*
;

/*
 * Arguments.
 */

Argument:
  constant = ID_PART | variable = VID | logical = LogicalConstant | expression = ObjectExpression | string = STRING
;

ArgumentList:
  arguments += Argument (',' arguments += Argument)*
;

ObjectExpression:
  base = VID (calls += ObjectExpressionMethodInvocation)+
;

ObjectExpressionMethodInvocation:
  '.' methodName = ID_PART '(' (arguments = ArgumentList)? ')'
;

LogicalConstant:
  value=("TRUE" | "FALSE");

/* Terminal definitions. */

terminal VID returns ecore::EString : '$' ID_PART;

//terminal ID  returns ecore::EString : '^'?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*(('.')('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*)*;

terminal FULL_ID  returns ecore::EString  : ID_PART+ ('.' ID_PART*)+;

terminal ID_PART returns ecore::EString : '^'?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;

//terminal INT returns ecore::EInt: ('0'..'9')+;

terminal STRING :
      '"' ( '\\' . /* 'b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\' */ | !('\\'|'"') )* '"' |
      "'" ( '\\' . /* 'b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\' */ | !('\\'|"'") )* "'"
    ;

terminal ML_COMMENT : '/*' -> '*/';

terminal SL_COMMENT   : '//' !('\n'|'\r')* ('\r'? '\n')?;

terminal WS     : (' '|'\t'|'\r'|'\n')+;

terminal ANY_OTHER: .;
```

Listing A.1: Grammar definition for RefDstl

Appendix B

RefDstl Standard Library

```
/**
 * This file is the standard library of refactorings for the RefDstl system
 * when being used to refactor Java programs.
 *
 * Essentially these are CRUD operations over a model of Java programs.
 */

/*
 * CReate operations.
 */

/**
 * Creates a new class with no members and no superclass.
 *
 * $className - The name of the class to create.
 */
proc createClass($className)
  requires $utils.classDoesNotExist($className)
{
  create class $className
}

/**
 * Creates a new class with no members but with a given superclass.
 *
 * $className - The name of the class to create.
 * $superClassName - The name of the class' superclass.
 */
proc createClassWithSuper($className, $superClassName)
  requires forall $cls in $utils.allClasses()
          : ~$cls.getFullyQualifiedName().equals($className)
  requires exists $cls in $utils.allClasses()
          : $cls.getFullyQualifiedName().equals($superClassName)
{
  create class $className with superclass $superClassName
}

/**
 * Creates a new field in a class.
 *
 * $fieldName - The name of the field to create.
 * $className - The fully qualified name of the class where the field is to be
    created.
 * $typeName - The type of the field.
 */
proc createField($fieldName, $accessor, $className, $typeName)
  requires exists $cls in $utils.allClasses()
               : $cls.getFullyQualifiedName().equals($className)
  requires exists $cls in $utils.allClasses()
               : $cls.getFullyQualifiedName().equals($className)
                 -> forall $fld in $cls.getFields()
                      : ~$fld.getName().equals($fieldName)
{
  create field $fieldName in class $className of type $typeName
}

/**
 * Creates a new static field in a class.
 *
 * $fieldName - The name of the field to create.
 * $className - The fully qualified name of the class where the field is to be
    created.
 * $typeName - The type of the field.
 */
proc createStaticField($fieldName, $accessor, $className, $typeName)
  requires exists $cls in $utils.allClasses()
               : $cls.getFullyQualifiedName().equals($className)
  requires exists $cls in $utils.allClasses()
               : $cls.getFullyQualifiedName().equals($className)
                 -> forall $fld in $cls.getFields()
                      : ~$fld.getName().equals($fieldName)
{
  create static field $fieldName in class $className of type $typeName
}

/**
 * Creates a void method in a class with the given argument list.
 *
 * $methodName - The name of the method to create.
 * $className - The fully qualified name of the class where the method is to be
     created.
 * $arguments - The arguments to the method.
 * $returnType - The type of the value returned by the method.
 */
proc createMethod($methodName, $className, $arguments, $returnType)
  requires exists $cls in $utils.allClasses()
               : $cls.getFullyQualifiedName().equals($className)
  requires forall $cls in $utils.allClasses()
               : $cls.getFullyQualifiedName().equals($className)
             -> (forall $mtd in $cls.getMethods()
                  : ~$utils.signatureMatches($mtd, $arguments))
```

```
{
  create public method $methodName in class $className /*taking [ $arguments ]
    */ returning $returnType
}

/**
 * Creates a void method in a class with the given argument list.
 *
 * $methodName - The name of the method to create.
 * $className - The fully qualified name of the class where the method is to be
     created.
 * $arguments - The arguments to the method.
 */
proc createVoidMethod($methodName, $className, $arguments)
  requires exists $cls in $utils.allClasses()
               : $cls.getFullyQualifiedName().equals($className)
  requires forall $cls in $utils.allClasses()
               : $cls.getFullyQualifiedName().equals($className)
                  -> (forall $mtd in $cls.getMethods()
                         : ~$utils.signatureMatches($mtd, $arguments))
{
  create public method $methodName in class $className /*taking [ $arguments ]
      */
}

/**
 * Update operations.
 */

 /**
  * Renames a class.
  *
  * $oldName - The current name of the class.
  * $newName - The new name of the class.
  */
 proc renameClass($oldName, $newName)
  requires forall $cls in $utils.allClasses()
               : ~$cls.getFullyQualifiedName().equals($newName)
 {
  rename class $oldName to $newName
 }

 /**
  * Renames a field.
  *
  * $oldName - The current name of the field.
  * $newName - The new name for the field.
  * $className - The name of the class where the field is declared.
  */
 proc renameField($oldName, $newName, $className)
  requires forall $cls in $utils.allClasses()
               : $cls.getFullyQualifiedName().equals($className)
```

```
                  -> (forall $fld in $cls.getFields()
                           : ~$fld.getIdentifier().equals($newName))
{
 rename field $oldName to $newName in class $className
}

/**
 * Renames a method.
 *
 * $oldName - The current name for the method.
 * $newName - The desired new name for the method.
 * $className - The name of the class where the method is currently located.
 */
proc renameMethod($oldName, $newName, $className)
 requires forall $cls in $utils.allClasses()
               : $cls.getFullyQualifiedName().equals($className)
                  -> (forall $mtd in $cls.getMethods()
                          : ~$mtd.getName().equals(newName))
{
 rename method $oldName to $newName in class $className
}

/**
 * Changes the type of a field.
 *
 * $fieldName - The name of the field whose type is to change.
 * $newType - The new type for the field.
 * $className - The name of the class where the field is declared.
 */
proc changeFieldType($fieldName, $newType, $className)
 requires forall $cls in $utils.allClasses()
               : $cls.getFullyQualifiedName().equals($className)
                  -> (forall $fld in $cls.getFields()
                          : $fld.getIdentifier().equals($fieldName)
                             -> $fld.eCrossReferences().isEmpty())
{
 change type of field $fieldName in class $className to $newType
}

/**
 * Changes the return type of a method.
 *
 * $methodName - The name of the method to change.
 * $newType - The new return type for the method.
 * $className - The name of the class where the method is defined.
 */
proc changeMethodType($methodName, $newType, $className)
 requires forall $cls in $utils.allClasses()
               : $cls.getFullyQualifiedName().equals($className)
                  -> forall $mtd in $cls.getMethods()
                         : $mtd.getName().equals($methodName)
                            -> $mtd.eCrossReferences().isEmpty()
```

```
{
  change type of method $methodName in class $className to $newType
}

/**
 * Changes the access modifier on a method to public.
 *
 * $methodName - The name of the method to be given the public access modifier.
 * $args - The arguments that form the method's signature.
 * $className - The name of the class where the method is defined.
 */
proc makeMethodPublic($methodName, $args, $className)
  requires TRUE
{
  make method $methodName in class $className public
}

/**
 * Changes the access modifier on a method to private.
 *
 * $methodName - The name of the method to be given the private access modifier
    .
 * $args - The arguments that form the method's signature.
 * $className - The name of the class where the method is defined.
 */
proc makeMethodPrivate($methodName, $args, $className)
  requires forall $cls in $utils.allClasses()
                : $cls.getFullyQualifiedName().equals($className)
                  -> forall $mtd in $cls.getMethods() :
                        $mtd.getName().equals($methodName) -> $mtd.
    eCrossReferences().isEmpty()
{
  make method $methodName in class $className private
}

/**
 * Changes the access modifier on a method to protected.
 *
 * $methodName - The name of the method to be given the protected access
    modifier.
 * $args - The arguments that form the method's signature.
 * $className - The name of the class where the method is defined.
 */
proc makeMethodProtected($methodName, $args, $className)
  requires forall $cls in $utils.allClasses()
                : $cls.getFullyQualifiedName().equals($className)
                  -> forall $mtd in $cls.getMethods() :
                        $mtd.getName().equals($methodName) -> $mtd.
    eCrossReferences().isEmpty()
{
  make method $methodName in class $className protected
}
```

```

/**
 * Changes the access modifier on a method to default.
 *
 * $methodName - The name of the method to be given the default access modifier
    .
 * $args - The arguments that form the method's signature.
 * $className - The name of the class where the method is defined.
 */
proc makeMethodDefault($methodName, $args, $className)
  requires forall $cls in $utils.allClasses()
              : $cls.getFullyQualifiedName().equals($className)
                -> forall $mtd in $cls.getMethods() :
                      $mtd.getName().equals($methodName) -> $mtd.
    eCrossReferences().isEmpty()
{
  make method $methodName in class $className default
}

/**
 * Changes the access modifier on a field to public.
 *
 * $fieldName - The name of the field to become public.
 * $className - The name of the class where the field is located.
 */
proc makeFieldPublic($fieldName, $className)
  requires TRUE
{
  make field $fieldName in class $className public
}

/**
 * Changes the access modifier on a field to private.
 *
 * $fieldName - The name of the field to be marked as private.
 * $className - The name of the class where the field is declared.
 */
proc makeFieldPrivate($fieldName, $className)
  requires forall $cls in $utils.allClasses()
              : $cls.getFullyQualifiedName().equals($className)
                -> forall $fld in $cls.getFields() :
                      $fld.getName().equals($fieldName) -> $fld.
    eCrossReferences().isEmpty()
{
  make field $fieldName in class $className private
}

/**
 * Changes the access modifier on a field to protected.
 *
 * $fieldName - The name of the field which should have its access modifier
    changed.
 * $className - The name of the class where the field is located.
 */
proc makeFieldProtected($fieldName, $className)
  requires forall $cls in $utils.allClasses()
                : $cls.getFullyQualifiedName().equals($className)
                  -> forall $fld in $cls.getFields() :
                          $fld.getName().equals($fieldName) -> $fld.
    eCrossReferences().isEmpty()
{
  make field $fieldName in class $className protected
}

/**
 * Changes the modifier on a field to default access.
 *
 * $fieldName - The field to make default.
 * $className - The name of the class where the field is declared.
 */
proc makeFieldDefault($fieldName, $className)
  requires forall $cls in $utils.allClasses()
                : $cls.getFullyQualifiedName().equals($className)
                  -> forall $fld in $cls.getFields() :
                          $fld.getName().equals($fieldName) -> $fld.
    eCrossReferences().isEmpty()
{
  make field $fieldName in class $className default
}

/**
 * Adds an argument to a method.
 *
 * $argName - The name for the new argument.
 * $argType - The type of the new argument.
 * $methodName - The name of the method where the new argument is to be added.
 * $className - The name of the class where the method is located.
 */
proc addMethodArgument($argName, $argType, $methodName, $className)
  requires forall $cls in $utils.allClasses()
                : $cls.getFullyQualifiedName().equals($className)
                  -> forall $mtd in $cls.getMethods()
                          : $mtd.getName().equals($methodName)
                            -> forall $p in $mtd.getParameters()
                                    : ~$p.getName().equals($argName)
{
  add argument $argType $argName to method $methodName in class $className
}

/**
 * Removes an argument from a method.
 *
 * $argName - The name of the argument to delete.
 * $methodName - The name of the method to delete.
 * $className - The name of the class where the method is currently located.
 */
proc deleteMethodArgument($argName, $methodName, $className)
  requires forall $cls in $utils.allClasses()
               : $cls.getFullyQualifiedName().equals($className)
                  -> forall $mtd in $cls.getMethods()
                        : $mtd.getName().equals($methodName)
                           -> forall $p in $mtd.getParameters()
                                 : $p.getName().equals($argName)
                                    -> $p.eCrossReferences().isEmpty()
{
  delete argument $argName from method $methodName in class $className
}

/**
 * Reorders the arguments to a method by moving one argument in front of the
   other.
 *
 * $methodName - The name of the method that should have its arguments
   rearranged.
 * $predecessor - The argument that should be closest to the head in the list
   of arguments.
 * $successor - The argument that should be closest to the tail in the list of
   arguments.
 * $className - The class where the method is currently located.
 */
proc reorderMethodArguments($methodName, $predecessor, $successor, $className)
  requires TRUE
{
  place argument $predecessor in method $methodName with args [ ARTYPE ARGNAME
   ] before $successor in class $className
}

/**
 * Changes the superclass that a subclass currently extends.
 *
 * $subclass - The class whose parent class is to be changed.
 * $superclass - The new class that $subclass extends.
 */
proc changeSuperclass($subclass, $superclass)
  requires forall $cls in $utils.allClasses()
               : $cls.getFullyQualifiedName().equals($className)
                  -> $cls.eCrossReferences().isEmpty()
{
  class $subclass extends $superclass
}

/**
 * Moves a field one step up the inheritance hierarchy.
 *
 * $fieldName - The name of the field to move up the inheritance hierarchy.
 * $className - The name of the class where the field is currently located.
 */
proc moveFieldToSuperclass($fieldName, $className)
  requires forall $fld in $utils.findClass($className).getFields()
                : $fld.getName().equals($fieldName) -> ($fld.isUnused() \/ ~$fld.
    isPublic())
{
  move field $fieldName from $className up
}

/**
 * Moves a field one step down the inheritance hierarchy.
 *
 * $fieldName - The name of the field to move.
 * $className - The name of the class where the field is currently located.
 */
proc moveFieldToSubclass($fieldName, $className)
  requires forall $cls in $utils.allClasses()
                : $cls.getFullyQualifiedName().equals($className)
                   -> forall $fld in $cls.getFields() :
                         $fld.getName().equals($fieldName) -> $fld.
    eCrossReferences().isEmpty()
{
  move field $fieldName from $className down
}

/**
 * Moves a method from a class to its superclass.
 *
 * $methodName - The name of the method to move.
 * $className - The name of the class where the method is currently located.
 */
proc moveMethodToSuperclass($methodName, $className)
  requires $utils.findMethod($className, $methodName).isUnused() \/ $utils.
    findMethod($className, $methodName).isPublic()
{
  move method $methodName from $className up
}

/**
 * Moves a method from a class to its subclass.
 *
 * $methodName - The name of the method to move.
 * $className - The class where that method is currently located.
 */
proc moveMethodToSubclass($methodName, $className)
  requires forall $cls in $utils.allClasses()
                : $cls.getFullyQualifiedName().equals($className)
                   -> forall $mtd in $cls.getMethods() :
                         $mtd.getName().equals($methodName) -> $mtd.
    eCrossReferences().isEmpty()
{
  move method $methodName from $className down
}

 /**
  * Moves a field from one class to another.
  *
  * $fieldName - The name of the field to move.
  * $fromClass - The class where the field is currently located.
  * $toClass - The class where the field is to be moved.
  *
  *   requires $utils.findClass($fromClass).getField($fieldName).isUnused();
  *   requires ~$utils.fieldExists($toClass, $fieldName)
  */
proc moveField($fieldName, $fromClass, $toClass)
  requires ~$utils.fieldExists($toClass, $fieldName)
{
  move field $fieldName from $fromClass to $toClass
}

/**
 * Moves a method by name from one class to another.
 *
 * Note: ALL methods with the name will be moved. This does not take the
 * remainder of the method signature into account.
 *
 * $methodName - The name of the method to move.
 * $fromClass - The class where the method is currently located.
 * $toClass - The class where the method is currently located.
 */
proc moveMethod($methodName, $fromClass, $toClass)
  requires $utils.findClass($fromClass).getMethod($methodName).isUnused()
  requires ~$utils.methodExists($toClass, $methodName)
{
  move method $methodName from $fromClass to $toClass
}

/*
 * Delete operations.
 */

/**
 * Deletes a class by name.
 *
 * $className - The name of the class to delete from the project.
 */
 proc deleteClass($className)
   requires forall $cls in $utils.allClasses() :
      (forall $mtd in $cls.getMethods() : $mtd.eCrossReferences().isEmpty())
      /\ (forall $fld in $cls.getFields() : $fld.eCrossReferences().isEmpty())
 {
  delete class $className
 }

```

```
/**
 * Deletes an unused field from a class.
 *
 * $fieldName - The name of the field to delete.
 * $className - The name of the class where the field is declared.
 */
proc deleteField($fieldName, $className)
 requires $utils.findField($className, $fieldName).isUnused()
{
 delete field $fieldName in class $className
}

/**
 * Deletes an unused method from a class.
 *
 * $methodName - The name of the method to delete.
 * $className - The name of the class to delete.
 */
proc deleteMethod($methodName, $className)
 requires $utils.findMethod($className, $methodName).isUnused()
{
 delete method $methodName in class $className
}
```

Listing B.1: Standard Library for RefDstl

Bibliography

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition.

Al-Mutawa, H. A., Dietrich, J., Marsland, S., and McCartin, C. (2014). On the Shape of Circular Dependencies in Java Programs. In *Australasian Software Engineering Conference*, pages 48–57, Sydney. IEEE.

Arcelli, F., Mangiacavalli, M., Pochiero, D., and Zanoni, M. (2015). On Experimenting Refactoring Tools to Remove Code Smells. In *Scientific Workshop Proceedings of the International Conference on Agile Software Development*, Helsinki. ACM.

Barker, R. and Tempero, E. (2007). A Large-Scale Empirical Comparison of Object-Oriented Cohesion Metrics. In *Asia-Pacific Software Engineering Conference*, pages 414–421, Aichi. IEEE.

Beck, K. (1999). *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Boston, 1st edition.

Bézivin, J. (2005). On the Unification Power of Models. *Software & Systems Modeling*, 4(2):171–188.

Bieman, J. and Kang, B. K. (1995). Cohesion and Reuse in an Object-Oriented System. In *Symposium on Software Reusability*, pages 259–262, New York. ACM.

Blackburn, S. M., Garner, R., Hoffmann, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., and Stefanovi, D. (2006). The DaCapo Benchmarks : Java Benchmarking Development and Analysis. In *Object-Oriented Programming Systems, Languages, and Applications*, pages 169–190, Portland. ACM.

Boehm, B. W. (1988). A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61–72.

Brett, D., Digg, D., Garcia, K., and Marinov, D. (2007). Automated Testing of Refactoring Engines. In *European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, pages 9–10, Cavtat. ACM.

Bruneton, E., Lenglet, R., and Coupaye, T. (2002). ASM: A Code Manipulation Tool to Implement Adaptable Systems. In *Adaptable and Extensible Component Systems*.

Carvalho Júnior, A., Silva, L., and Cornélio, M. (2007). Using CafeOBJ to Mechanise Refactoring Proofs and Application. *Electronic Notes in Theoretical Computer Science*, 184:39–61.

Cassell, K., Andreae, P., Groves, L., and Noble, J. (2009). Towards Automating Class-Splitting Using Betweenness Clustering. In *International Conference on Automated Software Engineering*, pages 595–599, Auckland. ACM/IEEE.

Chaparro, O., Bavota, G., Marcus, A., and Penta, M. D. (2014). On the Impact of Refactoring Operations on Code Quality Metrics. In *International Conference on Software Maintenance and Evolution*, pages 456–460, Victoria. IEEE.

Chidamber, S. R. and Kemerer, C. F. (1991). Towards a Metrics Suite for Object-Oriented Design. In *Object Oriented Programming Systems Languages and Applications*, pages 197–211, Phoenix. ACM.

Chidamber, S. R. and Kemerer, C. F. (1994). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. (2003). The Maude 2.0 System. *Rewriting Techniques and Applications*, LNCS 2706:76–87.

De Roover, C., Lammel, R., and Pek, E. (2013). Multi-Dimensional Exploration of API Usage. In *International Conference on Program Comprehension*, pages 152–161, San Francisco. IEEE.

Dig, D., Comertoglu, C., Marinov, D., and Johnson, R. (2006). Automated Detection of Refactorings in Evolving Components. In *European Conference on Object-Oriented Programming*, chapter 24, pages 404–428. Springer, Berlin.

Fokaefs, M. and Tsantalis, N. (2011). JDeodorant: Identification and Application of Extract Class Refactorings. In *International Conference on Software Engineering*, pages 1037–1039, Honolulu. ACM.

Fontana, F. A., Ferme, V., and Spinelli, S. (2012). Investigating the Impact of Code Smells Debt on Quality Code Evaluation. In *International Workshop on Managing Technical Debt*, pages 15–22, Zurich. IEEE.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston.

Garrido, A. and Meseguer, J. (2006). Formal Specification and Verification of Java Refactorings. In *International Workshop on Source Code Analysis and Manipulation*, pages 165–174, Philadelphia. IEEE.

Ge, X. and Murphy-Hill, E. (2014). Manual Refactoring Changes with Automated Refactoring Validation. In *International Conference on Software Engineering*, pages 1095–1105, Hyderabad. ACM.

Girvan, M. and Newman, M. E. J. (2002). Community Structure in Social and Biological Networks. *National Academy of Sciences of the United States of America*, 99(12):7821–7826.

Glass, R. (2001). Frequently Forgotten Fundamental Facts about Software Engineering. *IEEE Software*, 18(3):112–111.

Griffith, I., Reimanis, D., Izurieta, C., Codabux, Z., Deo, A., and Williams, B. (2014). The Correspondence Between Software Quality Models and Technical Debt Estimation Approaches. In *International Workshop on Managing Technical Debt*, pages 19–26, Victoria. IEEE.

Harman, M. and Clark, J. (2004). Metrics are Fitness Functions Too. In *International Software Metrics Symposium*, pages 58–69, Chicago. IEEE.

Henkel, J. and Diwan, A. (2005). CatchUp! Capturing and Replaying Refactorings to Support API Evolution. *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 0–9.

Hitz, M. and Montazeri, B. (1995). Measuring Coupling and Cohesion In Object-Oriented Systems. *Angewandte Informatik*, 50:1–10.

Hoare, C. A. R. (1969). An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580.

Jastram, M. (2014). *Rodin: User's Handbook*. CreateSpace Independent Publishing Platform.

Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. (2008). ATL: A Model Transformation Tool. *Science of Computer Programming*, 72(1-2):31–39.

Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., and Völkel, S. (2009). Design Guidelines for Domain Specific Languages. In *Workshop on Domain-Specific Modeling at Object Oriented Systems and Languages*, Orlando. ACM.

Lanza, M., Marinescu, R., and Ducasse, S. (2005). *Object-Oriented Metrics in Practice*. Springer.

Leavens, G. T., Baker, A. L., and Ruby, C. (1999). JML: A Notation for Detailed Design. In Kilov, H., Rumpe, B., and Harvey, W., editors, *Behavioural Specifications for Business and Systems*, chapter 12, pages 175–188. Kluwer.

Lehman, M. and Fernández-Ramil, J. C. (2006). Software Evolution. In Madhavji, N. H., Fernández-Ramil, J., and Perry, D. E., editors, *Software Evolution and Feedback: Theory and Practice*, chapter 1, pages 7–37. John Wiley & Sons.

Lincke, R., Lundberg, J., and Löwe, W. (2008). Comparing Software Metrics Tools. In *International Symposium on Software Testing and Analysis*, pages 131–142, Seattle. ACM.

Lorenz, M. and Kidd, J. (1994). *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall.

Marinescu, R. (2004). Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. In *International Conference on Software Maintenance*, pages 350–359, Chicago. IEEE.

Martin, R. C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall.

McCabe, T. J. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320.

McQuillan, J. A. (2011). *Using Model Driven Engineering to Reliably Automate the Measurement of Object-Oriented Software, PhD Dissertation*. Maynooth University.

Meneely, A., Smith, B., and Williams, L. (2012). Validating Software Metrics: A Spectrum of Philosophies. *ACM Transactions on Software Engineering and Methodology*, 21(4):1–28.

Mens, T. and Tourwe, T. (2004). A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139.

Mens, T., Van Eetvelde, N., Demeyer, S., and Janssens, D. (2005). Formalizing Refactorings with Graph Transformations. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(4):247–276.

Metsker, S. J. and Wake, W. C. (2006). *Design Patterns in Java*. Addison-Wesley, 2nd edition.

Meyer, B. (1997). *Object-Oriented Software Construction*. Prentice Hall, 2nd edition.

Milne, I. and Rowe, G. (2002). Difficulties in Learning and Teaching Programming - Views of Students and Tutors. *Education and Information Technologies*, 7(1):55–66.

Moghadam, I. H. and Ó Cinnéide, M. (2011). Code-Imp: A Tool for Automated Search-Based Refactoring. In *Workshop on Refactoring Tools*, pages 41–44, Honolulu. ACM.

Muller, P.-A., Fondement, F., Baudry, B., and Combemale, B. (2012). Modeling modeling modeling. *Software & Systems Modeling*, 11(3):347–359.

Murphy-Hill, E. and Black, A. P. (2008). Seven Habits of a Highly Effective Smell Detector. In *International Workshop on Recommendation Systems for Software Engineering*, pages 36–40, Atlanta. ACM.

Murphy-Hill, E., Parnin, C., and Black, A. P. (2012). How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering*, 38(1):5–18.

Myers, G. J., Sandler, C., and Badgett, T. (2011). *The Art of Software Testing*. Wiley Publishing, 3rd edition.

Nastov, B. (2013). *Grammar and Graphical Concrete Syntaxes Generator Assistant for Domain Specific Modelling Languages, M.Sc. Dissertation*. Montpellier II.

O'Keeffe, M. and Ó Cinnéide, M. (2003). A Stochastic Approach to Automated Design Improvement. In *Principles and Practice of Programming in Java*, pages 16–18, Kilkenny. ACM.

O'Keeffe, M. and Ó Cinnéide, M. (2006). Search-Based Software Maintenance. In *European Conference on Software Maintenance and Reengineering*, Bari. IEEE.

O'Keeffe, M. and Ó Cinnéide, M. (2008a). Search-Based Refactoring : An Empirical Study. *Journal of Software Maintenance*, 20(5):345–364.

O'Keeffe, M. and Ó Cinnéide, M. (2008b). Search-based Refactoring for Software Maintenance. *Journal of Systems and Software*, 81(4):502–516.

OMG (2015). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Technical report.

Opdyke, W. F. (1992). *Refactoring Object-Oriented Frameworks, Ph.D. Dissertation*. University of Illinois at Urbana-Champaign.

Overbey, J. L. and Johnson, R. E. (2011). Differential Precondition Checking: A Lightweight, Reusable Analysis for Refactoring Tools. In *Automated Software Engineering*, pages 303–312, Oread. IEEE/ACM.

Overbey, J. L., Johnson, R. E., and Munawar, H. (2016). Differential Precondition Checking: A Lightweight, Reusable Analysis for Refactoring Tools. *Automated Software Engineering*, 23(1):77–104.

Schaefer, I., Bettini, L., Bono, V., Damiani, F., and Tanzarella, N. (2010). Delta-oriented Programming of Software Product Lines. *Software Product Lines: Going Beyond*, 6287 LNCS:77–91.

Schneider, J.-G., Vasa, R., and Hoon, L. (2010). Do Metrics Help to Identify Refactoring? In *Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, pages 3–7, Antwerp. ACM.

Simon, F., Steinbruckner, F., and Lewerentz, C. (2001). Metrics Based Refactoring. In *Conference on Software Maintenance and Reengineering*, pages 30–38, Lisbon. IEEE.

Soares, G., Gheyi, R., Serey, D., and Massoni, T. (2010). Making Program Refactoring Safer. *IEEE Software*, 27(4):52–57.

Stachowiak, H. (1973). *Allgemeine Modelltheorie*. Springer.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley, 2nd edition.

Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., and Noble, J. (2010). The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *Asia Pacific Software Engineering Conference*, pages 336–345, Sydney. IEEE.

Thompson, S. (2011). *Haskell: The Craft of Functional Programming*. Addison-Wesley, 3rd edition.

Tokuda, L. (2001). Evolving Object-Oriented Designs with Refactorings. *Automated Software Engineering*, 8:89–120.

Völter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L., Visser, E., and Wachsmuth, G. (2013). *DSL Engineering Designing, Implementing and Using Domain-Specific Languages*. CreateSpace Independent Publishing Platform.