A Model for the Application of Arbitrary Object-Oriented Refactorings

by

Grant Gayed, B. C. S.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science

School of Computer Science
Carleton University
Ottawa, Ontario
July 29, 2002

© copyright
2002, Grant Gayed
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-79756-2
The undersigned recommend to the Faculty of Graduate Studies
and Research acceptance of the thesis

“A Model for the Application of Arbitrary Object-Oriented
Refactorings”

submitted by

Grant Gayed, B. C. S.

in partial fulfilment of the requirements for
the degree of Master of Computer Science

Signature of Thesis Supervisor

Signature of Director, School of Computer Science

Carleton University
July 29, 2002
Abstract

Refactoring is the act of changing an object-oriented program’s implementation such that its functional behaviour is not affected. Developers use it routinely to evolve the designs of programs into improved designs that better meet their needs. However there exists a risk of introducing new bugs to a program while applying a refactoring. This risk has recently been countered by the creation of tools that can apply selected refactorings to programs in an automated manner that is guaranteed to not introduce new problems. The shortcoming of these tools is their inability to apply refactorings for which they do not have previous knowledge. This thesis addresses this shortcoming. This thesis specifies a refactoring model and expression syntax that can be used to define any valid refactoring. Consequently, these facilities enable the development of refactoring tools that can be used to repeatedly apply any valid object-oriented refactoring to programs in an automated manner.
# Table of Contents

Abstract .................................................................................................................. iii

Table of Contents .................................................................................................... iv

List of Figures .......................................................................................................... vi

List of Appendices .................................................................................................... vii

Chapter 1. Introduction ........................................................................................ 1
  1.1 Problem ........................................................................................................... 1
  1.2 Motivation for Refactoring ............................................................................ 3
  1.3 Goals .............................................................................................................. 7
  1.4 Contributions .................................................................................................. 8
  1.5 Outline ........................................................................................................... 9

Chapter 2. Related Work ....................................................................................... 10
  2.1 Related Refactoring Work ............................................................................ 10
    2.1.1 Summary of Refactoring Work ............................................................... 28
  2.2 Related Object Model Expression Work ..................................................... 30
    2.2.1 Summary of Object Model Expression Work ...................................... 37

Chapter 3. Refactoring Model ............................................................................. 38
  3.1 Model ............................................................................................................. 38
    3.1.1 Actor Model ............................................................................................ 39
    3.1.2 Operation Model ...................................................................................... 42
      3.1.2.1 Operation Names .............................................................................. 45
      3.1.2.2 Actor Declarations .......................................................................... 45
      3.1.2.3 Actor Requirements ........................................................................ 45
      3.1.2.4 Refactorings ..................................................................................... 47
      3.1.2.5 Assertions ......................................................................................... 47
      3.1.2.6 Computations ................................................................................... 48
      3.1.2.7 Operation Invocations ................................................................... 49
      3.1.2.8 Assignments ..................................................................................... 50
      3.1.2.9 Iteration ............................................................................................ 51
      3.1.2.10 Conditionals .................................................................................. 51
      3.1.2.11 Primitive Operations ................................................................... 52
    3.1.3 Execution Model ....................................................................................... 54
  3.2 Expression of Refactorings ............................................................................ 55
    3.2.1 Expression of Actor Declarations ....................................................... 56
    3.2.2 Expression of the Operation Model ..................................................... 57
  3.3 Summary ......................................................................................................... 66
## List of Figures

| Figure 1. | Opdyke's low-level refactorings | 14 |
| Figure 2. | Narrative Description of the Rename Method refactoring defined in terms of the FAMOOS refactoring model | 21 |
| Figure 3. | Summary of reviewed refactoring studies | 31 |
| Figure 4. | Example CDIF snippets for hypothetical FAMIX class \( \text{Point} \) | 34 |
| Figure 5. | The GXL expression of a very simple program graph | 36 |
| Figure 6. | The primary entities and relationships in the utilized actor model | 41 |
| Figure 7. | The actor model's entity attributes | 43 |
| Figure 8. | DTD for the primary actor entity types | 58 |
| Figure 9. | DTD for the Number and String entity types | 60 |
| Figure 10. | DTD for all aggregate entity types | 61 |
| Figure 11. | DTD for all operation model entities | 64 |
| Figure 12. | Expression of the \( \text{AddParameter} \) refactoring | 73 |
| Figure 13. | Expression of the \( \text{CanAddParameterToMethod} \) assertion | 75 |
| Figure 14. | Expression of the \( \text{ExtractBody} \) refactoring | 86 |
| Figure 15. | Expression of the \( \text{CanExtractBlock} \) assertion | 92 |
| Figure 16. | Operation expression start element interactions | 98 |
| Figure 17. | Operation expression end element interactions | 99 |
| Figure 18. | Operation application interactions | 100 |
| Figure 19. | Operation application object model diagram | 101 |
| Figure 20. | Screen snapshot of the developed refactoring tool | 103 |
| Figure 21. | Implementation of the \( \text{CreateClass} \) primitive refactoring for Java | 105 |
List of Appendices

Appendix A. Refactoring Model Data Type Definition .................. 120
Chapter 1

Introduction

For any given program functionality there exists a vast array of potential underlying implementations. From the perspective of the program’s end user all of these implementations are of equal value, assuming that they also meet the user’s non-functional requirements, such as performance. To the end user, modifying a given program’s implementation such that its functional behaviour is not changed at all seems like an exercise without benefit. However, from the perspective of the software developer, various program implementations that exhibit identical functional behaviours are typically not of equal value. Implementation characteristics such as modularity, flexibility and clarity can distinguish one implementation from another as being superior in terms of its usefulness to the developer. This notion of software implementation quality extending beyond its functional behaviour is what motivates the desire to transform a given program implementation into some other form whose functional behaviour will be equivalent. In object-oriented contexts this practice is called refactoring, which is defined as the act of modifying the implementation of a syntactically valid program such that its functional behaviour is not changed.

1.1 Problem

Refactoring has become an integral part of object-oriented development. Developers use it routinely to evolve the designs of existing programs into improved
designs that better meet their needs. However, while the benefits of improving program
design through the application of refactorings can be enormous, there exists an inherent
risk of introducing new bugs to programs during the refactoring application process. This
risk is magnified for complex refactorings that affect many different parts of programs.
The costs that are associated with introducing new problems to existing software are
often extremely high, especially in cases where the problem is subtle in nature and
consequently does not trigger a compilation error. As such, the benefits that can be
realized by applying refactorings to a program can be negated if new problems are
introduced in the process.

This risk has been countered by the creation of tools that can apply selected
refactorings to programs in an automated manner as dictated by a user. Assuming correct
tool implementation, use of these tools for applying desired refactorings is a way to
ensure that new problems are not introduced during the refactoring application process.
Such tools have proven to be an extremely valuable aid to developers needing to use
program refactorings that can be applied by a utilized tool. By guaranteeing the proper
application of selected refactorings, these tools allow developers to focus on the task of
introducing improved design to programs without being burdened by all of the
refactoring application details. However, the primary shortcoming of these automated
tools is that they are unable to apply refactorings for which they do not have previous
knowledge. While there does exist a set of refactorings whose use is relatively
ubiquitous, the range of conceivable refactorings is certainly not confined to this set.
Refactorings can be applied at various design levels, and complex refactorings will tend
to be more context-specific, thus making their use less widespread. Additionally, the
benefits of performing certain refactorings may only be desirable within specific
domains. For example, some refactorings are not applicable to all object-oriented
languages due to the available language constructs, such as the utilization of multiple
inheritance in C++. Given these considerations, it is clearly not possible to create a tool
that has a priori knowledge of how to automatically apply all possible refactorings that a
developer may wish to apply to software. The inability of existing refactoring tools to
support the automated application of arbitrary valid object-oriented refactorings is the
problem that is addressed by this thesis.

1.2 Motivation for Refactoring

By its nature, most software is in a state of ongoing change. This characterization
applies not only to programs under active development, but also to programs that are
considered to be complete and hence are being utilized by end users. Beyond the
application of bug fixes, additional motives for making modifications to a program after
its initial release include the addition of new required functionality to it and the
adaptation of it to operate with new hardware and/or software for which it was not
originally designed. Clearly, software maintenance is a necessary activity for most
programs that are being actively used. As Wilde et al. noted: “Any successful software
system will eventually enter a prolonged and costly maintenance phase, not as a
consequence of failure, but of success. If a system is successful, users will demand that it
be strengthened and updated.” [WMH93].
It can be advantageous for a program to have a design that lends itself well to modification and extension, which in object-oriented contexts means encapsulation of functionalities that are prone to change. In fact, the attribute that is often used to describe a good object-oriented design is its ability to accommodate change. This is especially the case for software that has reached the maintenance stage of its lifecycle. Unfortunately, it has been shown that the quality of a program’s design typically degrades throughout its maintenance lifetime. This is not surprising since program changes made during the maintenance stage are typically driven by new requirements that were not considered in the original design, and consequently may not be well accommodated in the implementation. Making such changes often requires the addition of program code that is forced to fit into an inappropriate design, and consequently makes the resulting implementation brittle in nature.

A recent study [EGKM01] demonstrated the decay of an observed telephone switch program over a fifteen-year span. The study defined decay to be an increase in the difficulty of applying changes to the switch program, with the difficulty measure quantified in terms of the following characteristics:

- The span of code changes required in order to apply a change.
- The breakdown of modularity.
- The introduction of a new problem while applying a change.

The studied telephone switch program was observed to decay significantly over time in terms of these measures.
As the difficulty of performing maintenance on a program increases, the costs associated with the maintenance increase accordingly. In the context of software maintenance, costs are generally determined by the amount of time and risk involved in making a change. In spite of the relatively static nature of software that has entered its maintenance phase, it has been shown that maintenance work can account for over 70% of the total development cost [LS80]. This serves as a testament to the difficulty that is involved in changing software whose design has degraded over time. To further demonstrate this point, it has been shown that the most expensive software maintenance errors made have usually involved changes to only a few lines of source code [Wie83]. Realities such as these illustrate the cost savings that can be realized by ensuring that a program has a reasonably good design before attempting to make even small changes to it.

The best approach for controlling a program's maintenance costs is to prevent its design from decaying over time. When applying changes, this is achieved by carefully evolving a program's design as needed in order to make it better able to accommodate additions for which it may not be initially suited. Performing this kind of design evolution is among the primary usage contexts of refactorings today. This two-step approach to applying program changes helps to maintain the soundness of the program's design and therefore helps to keep it from becoming brittle over time.

More recently, the use of refactoring has spread to the context of the development of software that has not yet reached its maintenance stage. Refactoring has enabled the creation of new development methodologies that deviate from the traditional waterfall
model in ways that reflect the realities of large-scale software projects. These methodologies change the ways that developers work in an attempt to facilitate the creation of better software while minimizing the amount of required effort wherever possible. The methodology with the most pronounced deviation from the waterfall model, and not coincidently also with the most reliance on the use of refactoring, is Extreme Programming (XP) [Bec99].

XP advocates an evolutionary approach to determining software design, rather than the traditional approach of attempting to engineer it upfront. The motivation for advocating this approach is real world experience that indicates that design work that is performed upfront is usually far from optimal, and as a result is limited in terms of its utility for guiding implementation. The XP methodology advocates the minimization of upfront design work, and consequently forces many design decisions to be deferred to implementation time as needed. While making design decisions during implementation has traditionally resulted in software design that is inappropriate given a program’s full set of functional requirements, XP relies heavily on refactoring to enable developers to evolve a program’s design at implementation time as new requirements dictate. This ultimately leads to software design that is well suited to the system being implemented since it was the implementation that drove the development of the design. If used throughout a program’s development lifecycle, XP essentially turns the maintenance phase of a program into a special case of its original implementation phase. Alternative methodologies such as the spiral model [Boe86] do not go as far as to trivialize the design
phase of development, but still rely on refactorings to enable a program's design to be evolved as needed.

In summary, there are great cost savings associated with having good program design. Refactoring has established itself as an extremely useful technique for evolving the designs of programs into new designs that are comparatively superior in nature. Refactoring is useful from the initial implementation of a program right through to its maintenance stage. The primary caveat to keep in mind is that the improper application of refactorings, which can occur in contexts where refactoring of an application is manually performed, can negate these stated benefits.

1.3 Goals

The primary goal of this thesis is to specify a means by which developers can define and repeatedly apply any valid object-oriented refactoring to a program in an automated manner. This goal is composed of the following sub-goals:

- Specify a refactoring model: This is a formal set of language-independent refactoring definition concepts that can be used to fully describe any valid refactoring. The purpose of the model is to act as a common denominator between a developer that defines a refactoring and a tool that is subsequently used to apply the refactoring to existing software. It should be noted that while the refactoring model should be language-independent, it should still be usable for defining language-specific refactorings.
• Specify a means of using a tool to apply instances of the refactoring model in an automated manner: These are the facilities that are required in order to make the repeated application of a refactoring model instance possible.

This thesis will define one manifestation of each of these required sub-goals in order to fulfill its primary goal.

Additionally, the development of a refactoring tool that makes use of the research in this thesis would be of great benefit. Such a tool would provide some validation with respect to the appropriateness of the specified refactoring model and means of application. It would also demonstrate the applicability and usefulness of this research to real-world software developers.

1.4 Contributions

The contributions of this thesis are the development of each of the facilities that are required in order to meet the previously stated set of goals. These contributions are the following:

• A formal refactoring model, as described in the Goals section.

• A formal XML-based refactoring model expression syntax, which provides a means for developers to communicate defined refactoring model instances to tools that utilise the same refactoring model and expression syntax. This contribution is delivered in the form of an XML Document Type Definition (DTD) that formally states the rules of the syntax.
A prototype tool that is able to read instances of the refactoring model expressed according to the specified expression syntax and apply them to existing programs as specified by the refactoring model. The prototype tool is not completed to the extent of being usable for the automatic application of any defined refactoring model instance, but is able to apply defined refactorings whose attributes fall within certain constraints. This set of contributions provides a means for developers to realize the full benefits of applying refactorings to programs in a consistent automated manner without being constrained to using refactorings from a provided predefined set.

1.5 Outline

The next chapter provides an extensive background review on previous research that has been performed in the areas of object-oriented refactoring and object-oriented model expression. Following this, Chapter 3 formally describes the refactoring model that has been developed for modelling arbitrary refactorings, and the syntax that has been developed for expressing instances of this refactoring model. Chapter 4 provides concrete examples of two full refactorings that have been defined in terms of the developed refactoring model and expression syntax. It also provides cursory details of the developed prototype tool that can be used to apply these refactorings to existing programs. The final chapter reviews the research that has been performed in this thesis and suggests some opportunities for subsequent research that can be done in this area.
Chapter 2

Related Work

The notion of transforming programs and data structures from one representation to some other functionally equivalent representation existed prior to the initial efforts in object-oriented refactoring. For example, Griswold's research in restructuring of procedural and functional programs is in common practice. Additionally, existing database schemas have long been targets of data structure transformations. However, the topic of program transformations has taken on a much more significant role in object-oriented contexts as a result of the inherently structural nature of object-oriented programs [Opd92]. Since a major goal of this thesis is the development of a refactoring model, the majority of this chapter reviews research that address various aspects of the task of refactoring. Following this review is a short section that describes different approaches that have been used for concretely expressing information about object-oriented structures and refactorings in an open manner.

2.1 Related Refactoring Work

Griswold [Gri91] was among the first researchers to propose the use of automated restructurings as a means of reducing software maintenance costs. He observed that certain program structures were inherently better able to accommodate modifications than others, and consequently stated that transformations that introduced these favourable structures to programs under maintenance were of great value. In particular, he described
abstraction as being the primary cost reduction mechanism for enabling lower-risk
program modifications. He defined a set of behaviour preserving transformations that
could be applied to programs written in Scheme, many of which were particularly useful
for introducing abstraction. Program dependence graphs were used to formally prove
that all presented transformations would be behaviour preserving if applied properly
whenever certain preconditions for application held. Some example transformations and
their corresponding descriptions were:

- Abstract expression (expr-to-binding): This transformation encapsulated selected
  expression(s) into a new scope binding. The impact of isolating these expressions
  was that scopes containing alternative collections of expression(s) could then be
  easily substituted for those that were extracted. This resulting structure was
  similar to the Strategy [GHJV95] pattern that is now prevalent in object-oriented
  contexts.

- Extract function (extract-function): This transformation was similar to expr-to-
  binding, except that the extracted expression(s) were encapsulated into a new
  function that could be invoked from various places throughout the program.

- Scope-wide function replacement (scope-sub-call): This transformation replaced
  repeated sequences of code with function calls to an isolated function with the
  same code. This was typically used with extract-function in order to increase
  code reuse.

In addition to defining Scheme transformations, Griswold implemented a tool that could
apply them in an automated manner as directed by a user. While the context of his
research was slightly different from that of object-oriented refactoring, he clearly sought to accomplish the same goals as many of the subsequent refactoring studies.

Griswold’s research is interesting here because his chosen approach clearly had a great influence on many of the other studies that are reviewed in this section. However the inherently different nature of procedural and object-oriented programming languages limits the direct applicability of procedural language studies, such as [Wei84] and [Par79], to this thesis. Consequently the rest of this section specifically examines refactoring research that has been performed in the context of object-oriented software. The first three studies detail the efforts of others to define refactorings with proven behaviour preservation. The subsequent two reviews are similar in nature but specifically address the use of refactorings for the introduction of design patterns to software. Following this is an examination of a model that has been developed for representing refactorings, which matches one of the goals of this thesis. The final group of reviewed studies examines research related to the development of refactoring tools. These tools demonstrate both user-driven and automated/adaptive approaches to applying transformations to software.

Opdyke [Opd92] performed the first formal study on object-oriented refactoring. Opdyke defined a refactoring to be a “behaviour preserving program restructuring”, where the term “behaviour preserving” referred solely to functional behaviours of the restructured program, but not non-functional ones such as its resulting performance or memory usage. This is the refactoring definition that is used throughout most of the reviewed refactoring literature. A refactoring was specified to consist of precondition(s)
whose satisfaction was mandatory, and transformation steps whose subsequent application would perform the prescribed restructuring. Opdyke then defined twenty-six low-level refactorings, listed in Figure 1, and formally proved each to be behaviour preserving if properly applied. His proofs of behaviour preservation were based upon showing the preservation of seven syntactic and semantic properties of hypothetical code being acted upon. Finally, three composite refactorings related to framework manipulation were defined in terms of these low-level refactorings, thus making the composite refactorings behaviour preserving as well. The three composite refactorings were as follows:

- Create abstract superclass: Create a superclass that contains more general aspects of a given class.
- Create subclass: Create a subclass that contains more specialized aspects of a given class.
- Capture aggregations and components: Convert a relationship modeled using inheritance into an association.

While Opdyke’s research was performed in the context of C++ refactorings, most of his low-level refactorings are also applicable in other object-oriented languages such as Smalltalk and Java. By formally defining the preconditions and steps for applying these refactorings, Opdyke provided a basis for the definition of many more behaviour preserving composite refactorings in terms of these low-level ones.

There exists one primary shortcoming of the static refactoring definition approach taken by Opdyke. This shortcoming is that conservative estimations often must be made
Creating a Program Entity:
- create_empty_class
- create_member_variable
- create_member_function

Deleting a program entity:
- delete_unreferenced_class
- delete_unreferenced_variable
- delete_member_functions

Changing a program entity:
- 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

Moving a member variable:
- move_member_variable_to superclass
- move_member_variable_to subclass

Intermediate level refactorings:
- abstract_access_to_member_variable
- convert_code_segment_to_function
- move_class

Figure 1. Opdyke's low-level refactorings
when determining satisfaction of preconditions, and in particular the *IsOwned* condition. This condition is satisfied whenever all instances of a class are guaranteed be component objects of at most one composite class. Unfortunately determining this for all valid cases is difficult, so its criteria for satisfaction is made deliberately strict in order to ensure that satisfaction cannot occur in cases where it should not. This strictness leads to the constraint of many use cases in which the exclusivity condition is in fact satisfied but is not evaluated as such.

Roberts [Rob99] addressed this shortcoming a few years later. He proposed that increased knowledge of a program’s internal class relationships could be gathered dynamically by collecting information on its observed object interactions during a typical execution. Features of object-oriented languages such as polymorphism can make it difficult to accurately determine type of this information, especially in weakly typed languages like Smalltalk. If it is assumed that the dynamically collected information is relatively complete then it may be used to provide more detailed program analysis than a static approach can, and consequently more accurate answers to refactoring precondition queries.

The necessarily incomplete nature of observing program execution makes the use of dynamically collected information as the sole means of program analysis quite unadvisable. Most refactoring preconditions cannot ever be conclusively satisfied using dynamic analysis since the possibility always exists that some program paths were not exercised during the observed execution period. However, when combined with static analysis techniques, dynamic analysis can provide a more complete picture of what is
happening in a program at execution time. Roberts likens the "semi-proof" nature of
dynamic analysis to the way that developers typically work, since developers generally
rely on pre-existing knowledge of the system and personal intuition in order to make
decisions rather than always resorting to formal proofs. As such, he believes that
determining the satisfaction of conditions such as Opdyke's *IsOwned* condition should
not need to be unnecessarily strict since a developer should be able to leverage his/her
knowledge of the system reasonably well in order to aid in this determination. Roberts'
research is the only refactoring study that has not used solely static analysis techniques in
order to determine conditions for behaviour preservation. This is probably a result of the
comparatively simple and deterministic nature of using static analysis.

Werner [Wer99] formalised a language for defining composite refactorings. He
defined a Change Specification Language (CSL) whose commands could be invoked to
perform refactorings on programs that had been reverse-engineered to instances of his
developed Implementation Object Model (IOM). The IOM was sparse in nature,
consisting exclusively of nodes representing types and arcs representing attributes,
operations or associations. Each CSL command had both weak and strong sets of
associated preconditions in order to allow clients to evade certain constraints if they were
confident that application of a command would not adversely affect program behaviour.
Similar to Opdyke's approach, the CSL provided a set of low-level commands that
performed basic refactorings, and higher-level commands that performed more complex
refactorings by invoking the low-level ones. While many of these low-level commands
corresponded with low-level refactorings defined by Opdyke, there were also other
commands for performing transformations specifically related to the moving of responsibilities between classes. These more complex commands were DelegateOperation, ReclaimOperation, DelegateField and ReclaimField.

Werner’s research is quite interesting since his goal of formalizing a language for the definition of composite refactorings is similar to one of the goals of this thesis. Additionally, his ultimate implementation of a tool for automated application of these refactorings provides a concrete demonstration of its use. However, the utility of his presented approach is limited since it does not provide facilities beyond those for invoking provided CSL commands. Also limiting the approach’s potential scope is its lack of available CSL commands for manipulation of method bodies. These constraints reduce the types of refactorings that can be defined using the presented approach, and hence require the provision of additional facilities if the ability to define arbitrary refactorings is desired.

Ó Cinnéide [ÓCin00] defined an approach for using refactorings to introduce design patterns to programs where they were not previously used. The approach taken was to start by modeling each Gang of Four [GHJV95] design pattern as a composition of shared “mini-patterns”. These mini-patterns represented lower level patterns that were observed to recur within high-level design patterns. Each mini-pattern was then given an associated mini-refactoring that described the preconditions, behaviour-preserving transformations and post-conditions for restructuring existing code such that its structure would reflect the mini-pattern. Consequently, the refactorings required for restructuring specified code into a Gang of Four pattern were performed by successively applying each
of the component mini-refactorings, which is a similar approach to that taken by other studies that define high-level refactorings in terms of lower-level ones whose behaviour preservation has been shown.

The aspect of Ó Cinnéide's research that sets it apart from the other reviewed refactoring studies is its exclusive use of relatively high-level refactorings for achieving its goals. The six mini-transformations that were defined, with brief descriptions, were as follows:

- Abstraction: Adds an interface to a class.
- EncapsulateConstruction: Packages object creation statements into dedicated methods.
- AbstractAccess: Abstracts a class-to-class relationship to a class-to-interface relationship.
- PartialAbstraction: Constructs an abstract class from an existing class and introduces the consequently required extends relationship.
- Wrapper: Wraps an existing receiver class with another class.
- Delegation: Moves part of an existing class to a component class and introduces the consequently required delegation relationship.

This higher-level approach seems appropriate given the level of abstractions that characterize design patterns, and as a result a moderate level of success was ultimately reported. Additionally, a tool was developed that could be used to apply one of seven Gang of Four patterns to specified code.
Pieter van Winsen [Win96] also outlined an approach for using refactorings to introduce design patterns to existing programs. He chose to model object-oriented entities using the Fragment Model [Mei96], which represented all program entities (classes, methods, associations, etc.) uniformly as fragments connected to each other by edges communicating their respective roles. This fragment-role representation facilitated the mapping of one instance of the model to some other target instance if appropriate fragments were available in the source for fulfilling all required roles in the target. Similar to Ó Cinnéide’s approach, van Winsen began by developing Fragment Model representations of a few Gang of Four patterns that were to act as the targets of such mappings. A developer wishing to introduce one of these patterns to an existing program would then do so by using an existing tool to reverse-engineer the program into a Fragment Model representation, then defining mappings between selected fragments in the program and the target pattern’s required roles. Given these mappings, a hypothetical tool could then apply the appropriate Opdyke-defined refactorings to the source program in order to make its structure exhibit the target design pattern. While this study is interesting in that it shows a novel context in which partially automated refactorings would be useful, its minimal coverage of the methodology for transforming one Fragment Model instance to another limits its utility to this thesis.

Members of the FAMOOS project performed a feasibility study on the potential for defining refactorings that would be uniformly applicable across all object-oriented languages. This study has great relevance to the thesis since the development of a language-independent refactoring model is a primary goal. They developed a refactoring
model [TDDN00] whose role was to capture the common elements of object-oriented refactorings. This was accomplished by defining the model such that it essentially contained the intersection of the concepts contained in these languages without giving consideration to details such as abstract syntax trees and program flows within methods. Additionally, mappings were provided for relating some language-specific constructs to the model, such as the representation of Java Interfaces as abstract classes. While this overall approach did result in preserving the language-independent nature of the model, it constrained the model from representing some common refactorings such as ExtractMethod and MoveMethod.

The FAMOOS refactoring model utilised the FAMIX [DTS99] model for representing object-oriented entities involved in the refactoring, and expressed required preconditions and transformations to be applied to these entities in a narrative manner. These narrations detailed not only the steps to be performed, but also any language-specific interpretations that would be required for correct application of the refactoring. An example of their definition of refactoring Rename Method is shown in Figure 2. Tool support for applying predefined refactorings defined in terms of the model was subsequently built into the Moose [DLT00] refactoring tool, though no means of implementing new instances of the model for use by the tool was described.

This feasibility study demonstrates the interest that exists for the creation of a language-independent refactoring model. It provided a glimpse of some of the issues that are necessarily involved in such a task, such as the handling of differences between languages. However, the presented model did not address many of the requirements for
Rename Method (method, new name)

Description:
Renames method and all method definitions with the same signature in the same hierarchy. All invocations to all changes methods are changed to refer to the new name.

Language-Independent Preconditions:
The subclass hierarchies of the classes highest up in the superclass hierarchies of the class containing method do not already contain a method with a signature implied by the new name and the parameters of method

Language-Dependent Preconditions:
New name is a valid method name

Smalltalk-Specific Preconditions:
There exists no method with the same signature as method outside of the inheritance hierarchy of the class that contains method.

Java-Specific Preconditions:
When method is a constructor the refactoring cannot be applied unless in the context of a rename class refactoring.

Figure 2. Narrative Description of the Rename Method refactoring defined in terms of the FAMOOS refactoring model
allowing developers to define arbitrary refactorings for widespread application. One primary shortcoming of the model was the failure of its utilised FAMIX model to consider many code-level details, which prevented it from representing some very common refactorings. Another major shortcoming of the model was its lack of formalism. While the narrative description of the Rename Method refactoring communicated its required steps reasonably well, a developer wishing to implement it would need to write custom code for each target environment since no higher-level concepts are formally defined. This lack of formalism also inhibits the development of a higher-level syntax that could be used by developers to circumvent this customisation requirement. These shortcomings are addressed in the refactoring model that is presented in this thesis.

Fowler [FBBO99] recognized the existence of a set of common refactorings that are frequently used throughout the development process. He compiled seventy-two of these refactorings into a catalogue, and for each gave a narrative description of the required respective preconditions and transformation steps for manual application such that behaviour would be preserved. The classification of these refactorings as being common foreshadows their relatively low-level nature. Fowler acknowledged that while cataloguing the correct application of these refactorings was important, it was by no means exhaustive in terms of the refactorings that developers may wish to apply. He also discussed the importance of having automated tools to aid in the application of refactorings in order to eliminate the risk of introducing subtle new problems in the process.
Roberts, Brant and Johnson [RBJ97] implemented the first refactoring tool whose purpose was to perform the application of refactorings in an automated manner. The tool was essentially a Smalltalk development environment with built-in functions for performing predefined refactorings on selected program entities. Like the refactorings in Fowler’s catalogue, the refactoring functions provided in the tool were chosen based upon their common nature of usage. When the user invoked a refactoring, the tool’s refactoring function verified all required preconditions, and if satisfied, applied the refactoring as appropriate. This provided developers with a means of safely performing many common refactorings with minimal effort, and facilitated the subsequent development of more aggressive programming practices such as Extreme Programming.

In recent years refactoring support of a similar nature has appeared in various commercial development environments we well, such as Instantiations’ jFactor [jfactor] and the Eclipse Project [eclipse].

While refactoring tools can greatly aid in the application of refactorings, the potentially large scope of a given software project can make it difficult for a developer to identify the refactorings whose application would be most beneficial. Bowdidge [Bow95] defined a visualization technique called the Star Diagram whose specific purpose was to aid in the identification of refactoring opportunities. This was done to compliment the previously described research of Griswold. The diagram provided a compact abstract syntax tree view of all uses of a specific data structure in a Scheme program, thereby allowing a developer to see the detailed ways in which clients were utilizing a given entity. Although providing visuals at the abstract syntax tree level often
poses the threat of becoming overwhelming, the diagram’s node structuring conventions made it reasonably easy to scan large diagrams for characteristics such as duplicated code, shown as stack structures, or excessively long chains of reference. Bowdidge implemented a corresponding tool that could visualize Scheme programs as Star Diagrams and perform selected refactorings on the diagram nodes. The tool’s extended visualization mechanism allowed the user to selectively reduce the amount of information shown by specifying branches or node types for exclusion. Bowdidge also discussed the issues involved in creating Star Diagrams for representing programs written in other procedural languages such as C. Korman [Kor98] subsequently implemented a tool called Elbereth that visualized Java programs as Star Diagrams. While Elbereth gave the user the option to prune branches or node types from visualizations like Bowdidge’s tool, it was not able to perform automated refactorings on the nodes. This tool is a significant achievement nonetheless since it demonstrates the applicability of the Star diagram to object-oriented languages.

The Demeter group performed adaptive programming studies whose program transformation aspects have relevance to the study of refactoring. They used Class Dictionary Graphs (CDG) [LX93] to represent reverse-engineered systems, which like Werner’s IOM, consisted solely of nodes and edges. Its nodes represented either concrete classes that could be instantiated but not subclassed, or abstract classes. Its edges represented either construction edges that denoted references between objects, or alternation edges that denoted inheritance. Similar to Werner’s practice of manipulating
IOM graphs, Demeter group research performed program refactorings by manipulating Class Dictionary Graphs.

In his presentation of behaviour-preserving CDG refactorings, Bergstein [Ber94] used the term "object equivalent" to describe pairs of CDG instances that allowed for the same set of objects. He showed that a CDG transformation whose application resulted in an object equivalent CDG with respect to the original was behaviour preserving. Bergstein then defined five CDG-specific primitive transformations that exhibited this property. These object-equivalent transformations were:

- Deletion of Useless Alternation
- Addition of Useless Alternation
- Abstraction of Common Parts
- Distribution of Common Parts
- Part Replacement

In contrast, Hüsch [Hür95] defined sets of basic and composite CDG transformations that were not as closely tied to the model. Additionally, Hüsch defined transformations that could potentially be destructive, thereby giving the client increased freedom to make changes if deemed appropriate.

Another topic explored by the Demeter group was the use of static code metrics for detecting refactoring opportunities. They defined the Law of Demeter [Hol93] whose application imposed constraints on the types of objects to which a given class' instances would send messages. Specifically, the Law stated that instances of a class should only send messages to a given object if the object satisfied one of the following criteria:
• The object was self.

• The object was created by self.

• The object was an attribute of self.

• The object was received by self as a method argument.

The original goal of the Law was to provide an objective means for determining situations in which CDGs could be simplified. However, adherence to the Law was just as potentially valuable to those not utilizing these graphs since their direct class dependencies would typically be reduced as a result. A set of refactorings that could be used to apply the Law of Demeter to existing programs was presented in [LHR88]; these refactorings effectively added delegation methods to intermediate classes in message send chains in order to reduce improper direct access on the part of their clients. The Law of Demeter is interesting because it facilitates the complete automation of one specific refactoring from the point of problem detection right through to the actual program transformation. Many other criteria exist for objectively evaluating object-oriented programs, some of which are useful for defining additional refactorings whose application can be fully automated. The advantage of these automated approaches over developer-driven refactorings is their potential for applying conceived best implementation practices universally to programs with a minimal expenditure of effort. However these best practices must be chosen carefully so as to not complicate an existing design in the process. The Law of Demeter achieved this well by using the straightforward solution of introducing delegation when needed in order to reduce direct class dependencies.
Moore and Clement [MC96] outlined an algorithm for completely restructuring a given set of classes such that their resulting inheritance hierarchies would be optimal in terms of minimization of functional duplication. Their algorithm considered duplication down to the abstract syntax tree level, so common method bodies were often factored out in addition to methods and fields. Moore also developed a complimentary tool called Guru [Moo95] that applied this optimization to existing programs written in Self [US87]. Casais [Cas92] proposed a less drastic approach to class hierarchy optimization in which existing hierarchies were optimized in an incremental fashion through the introduction of abstractions that captured new commonalities. This approach had the advantage of performing less design changes than the algorithm of Moore and Clement, though the potential for introducing reuse optimizations was reduced as a result.

Kataoka, Ernst, Griswold and Notkin [KEGN01] described an approach for analysing software using invariants. The goal of this research was to provide a means by which a developer could define any set of criteria as being characteristics of a design problem requiring refactoring. A given program could then be easily validated with respect to these criteria in an automated manner, thus enabling the automatic detection of the developer’s conceived refactoring opportunities. For example, the condition indicating an opportunity to apply the RemoveParameter refactoring was defined as:

- \( p = \text{constant} \) \{ie.- the parameter’s received value is always the same constant\}
- \( p = f(a, b, \ldots) \) \{ie.- parameter \( p \) can be computed using the other parameters\}

where:

- \( p \) is a parameter
• f is a computable function

• a, b, … are either parameters or other variables in scope at the procedure entry

This approach is more flexible than the other automated techniques described since it allows a developer to customize the criteria for detecting code with “bad smells”, which in various contexts could be subjective. A similar but simpler means of defining custom code quality criteria is to analyze programs in terms of metrics, and define thresholds for these metrics beyond which a refactoring is required. Good compilations of object-oriented metrics include [CK94] and [LK94]. Metrics were incorporated into the Moose Refactoring Tool (“MOOSE/Metrics”) [DD99] that measured various program metrics taken from these two sources with the purpose of identifying either refactoring opportunities or recently applied refactorings.

2.1.1 Summary of Refactoring Work

Given the broad range of approaches that were taken by the reviewed studies, providing a concise summary that compares them meaningfully is not a trivial task. Ó Cinnéide suggested that two dimensions that can be used for classifying refactoring approaches, and their respective possible values, are:

1. Intended method of application:

   • Fully automated: A tool performs automatic large-scale analysis and/or refactoring across a given program.

   • Semi-automated: A tool performs analysis and/or refactoring on a specific part of a given program as specified by a client.
• Manual: No automation is utilized.

2. Proof of behaviour preservation:

• Formal: A proof of behaviour preservation based upon the formal semantics of the target is provided.

• Semi-formal: Some approach such as first-order predicate logic is used to support the argument, but it is not based strictly upon the semantics of the target.

• Narrative: No formal proof is provided, typically just a narrative justification.

Given the specific topic of this thesis, another appropriate dimension for classifying refactoring approaches is the means by which a client of an approach is able to define a new refactoring. Assigning this dimension one of the following possible values allows for the easy classification of the reviewed studies:

• Custom: A client can define any refactoring and/or analysis function for application.

• Composition: A client can define a refactoring and/or analysis function solely in terms of a predefined collection of provided functions.

• None: Definition of additional refactoring and/or analysis functions cannot be performed.

While these dimensions do not provide insight into the contents of the various studies, they provide a simple means by which the general approaches used by each study can be compared. Figure 3 summarizes the reviewed studies in terms of these
dimensions. Study dimensions for which a value could not be determined are assigned values of either "not applicable" or "not specified" as appropriate.

If the research in this thesis were to be classified according to these dimensions, it would be semi-automatic in nature with a client-determined means of behaviour preservation proof and a means of extension that most closely resembles custom. The clarification of the third dimension is made because custom refactorings necessarily must ultimately be defined in terms of lower-level functions in order to act upon a client.

The reviewed study whose shortcomings are most closely addressed by this thesis is the FAMOOS refactoring model, since the development of a refactoring model is one goal of this thesis. The refactoring model defined by this thesis will have the same set of base capabilities for defining refactorings as the FAMOOS model has, as well as additional facilities for dealing with the FAMOOS model's shortcomings. These shortcomings are:

- Its inability to model method body information beyond method invocations and field accesses.
- Its narrative nature for defining model instances.
- Its undefined means of extension.

2.2 Related Object Model Expression Work

The typical motivation for defining object model expression syntaxes is to enable the exchange of model instances between tools. As a result of this common motive, most object model expression syntaxes are concerned solely with the ability to communicate
<table>
<thead>
<tr>
<th>Study</th>
<th>Method of Application</th>
<th>Proof of Behaviour Preservation</th>
<th>Means of Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>Griswold [Gri91]</td>
<td>semi-auto</td>
<td>formal</td>
<td>composition</td>
</tr>
<tr>
<td>Opdyke [Odp92]</td>
<td>semi-auto</td>
<td>formal</td>
<td>composition</td>
</tr>
<tr>
<td>Roberts [Rob99]</td>
<td>semi-auto</td>
<td>semi-formal</td>
<td>composition</td>
</tr>
<tr>
<td>Werner [Wer99]</td>
<td>semi-auto</td>
<td>formal</td>
<td>composition</td>
</tr>
<tr>
<td>O' Cinnéide [ÓCin00]</td>
<td>semi-auto</td>
<td>semi-formal</td>
<td>none</td>
</tr>
<tr>
<td>Pieter van Winsen [Win96]</td>
<td>semi-auto</td>
<td>formal (Opdyke)</td>
<td>none</td>
</tr>
<tr>
<td>FAMOOS Refactoring Model [TDDN00]</td>
<td>semi-auto</td>
<td>narrative</td>
<td>not specified</td>
</tr>
<tr>
<td>Fowler [FBBO99]</td>
<td>manual</td>
<td>narrative</td>
<td>composition</td>
</tr>
<tr>
<td>Refactoring Browser [RBJ97]</td>
<td>semi-auto</td>
<td>formal (Opdyke)</td>
<td>composition</td>
</tr>
<tr>
<td>Bowdidge [Bow95]</td>
<td>semi-auto</td>
<td>formal</td>
<td>none</td>
</tr>
<tr>
<td>Bergstein [Ber94]</td>
<td>fully-auto</td>
<td>formal</td>
<td>composition</td>
</tr>
<tr>
<td>Hürsch [Hür95]</td>
<td>fully-auto</td>
<td>formal</td>
<td>composition</td>
</tr>
<tr>
<td>Law of Demeter [Hol93]</td>
<td>fully-auto</td>
<td>formal</td>
<td>none</td>
</tr>
<tr>
<td>Moore and Clement [MC96]</td>
<td>fully-auto</td>
<td>formal</td>
<td>none</td>
</tr>
<tr>
<td>Casais [Cas92]</td>
<td>fully-auto</td>
<td>formal</td>
<td>none</td>
</tr>
<tr>
<td>Kataoka, Ernst, Griswold and Notkin [KEGN01]</td>
<td>fully-auto</td>
<td>not applicable</td>
<td>custom</td>
</tr>
</tbody>
</table>

Figure 3. Summary of reviewed refactoring studies
static object model instances without capturing actions that are to be applied to them. While the model expression approaches taken by these studies are beneficial to explore, their usage context limits their direct applicability to this thesis. The one study which does consider the concrete expression of actions to be applied to object model instances is Werner’s CSL, which is reviewed first.

As was previously described, Werner formalized a Change Specification Language (CSL) for concretely expressing refactorings. This is the only well-known attempt at creating a language with the purpose of capturing object-oriented structure changes as opposed to simply modeling existing structures. Some example CSL commands that acted upon classes were:

- ADD CLASS ClassName
- DROP CLASS ClassName
- RENAME CLASS ClassName NEW NAME NewClassName

The syntax of the CSL has many similarities to SQL, which is a reflection of refactoring’s schema transformation roots. While this is an appropriate approach given the declarative nature of the language, it is not adequate for expressing refactorings in terms beyond those provided by the standard commands. Nonetheless, the CSL provides a straightforward means of expressing many common refactorings.

Various formats have been formalized for concretely expressing static object-oriented structures. The FAMOOS group defined a means of expressing instances of the FAMIX model as part of their FAMIX specification. The expression syntax is defined in terms of the CDIF [cdif] specification, which defines a generic syntax for expressing
instances of models via flat ASCII streams. The formal FAMIX CDIF declaration is available at [famixcdif]. Figure 4 shows example snippets from the CDIF expression of a hypothetical FAMIX class Point. An alternative expression format that is similar to that used by FAMIX but does not conform to the CDIF specification is the Bauhaus Resource Graph [CEKK00].

An alternative model expression approach that has been explored is the use of generic graph exchange formats. GXL [HW00] began as a format for exchanging object models between tools, but was later generalized to express graphs irrespective of their usage context. The graph types that GXL is able to model include typed, attributed, directed, and ordered, and hence any object-oriented model that is defined in terms of these types of graphs can be expressed. However it should be noted at this point that techniques for reverse engineering a given program to a graph representation is a separate issue that resides outside of the scope of GXL.

GXL graphs are concretely expressed using XML [xml] markup corresponding to the formal GXL data type definition. As such, a given graph expression can be validated both syntactically and semantically by any available XML parser. Figure 5 shows the GXL expression of a very simple program graph, from [HW00]. As can be inferred from the figure, GXL tags are defined solely in terms of generic graph concepts whose contents are to be interpreted as dictated by the usage context.
CDIF, SYNTAX "SYNTAX.1" "02.00.00", ENCODING "ENCODING.1" "02.00.00"
(:HEADER
  (:SUMMARY
    (ExporterName "sniff2famix")
    (ExporterVersion "1.1")
    (ExporterDate "1999/06/16")
    (ParsedSystemName "Point.shared")
    (ReificationLevel "3")
    (SourceLanguage "Java")
  )
)
(:META-MODEL
  (:SUBJECTAREAREFERENCE Foundation
    (:VERSIONNUMBER "01.00")
  )
  (:SUBJECTAREAREFERENCE FAMIX
    (:VERSIONNUMBER "2.0")
  )
)
(:MODEL
  (Class FM1
    (name "Point")
    (uniqueName "Point")
    (isAbstract -FALSE-)
    (sourceAnchor #[[file "Point.java" start 1 end 179]]#)
  )
  (Method FM3
    (name "getX")
    (belongsToClass "Point")
    (uniqueName "Point.getX()")
    (sourceAnchor #[[file "Point.java" start 113 end 116]]#)
    (accessControlQualifier "public")
    (hasClassScope -FALSE-)
    (signature "getX()")
    (isAbstract -FALSE-)
    (isConstructor -FALSE-)
    (declaredReturnType "int")
  )
  (Access FM5
    (accesses "Point._x")
    (accessedIn "Point.Point(int,int)")
  )
)
Figure 4. Example CDIF snippets for hypothetical FAMIX class *Point*
Figure 5. The GXL expression of a very simple program graph
2.2.1 Summary of Object Model Expression Work

Of the reviewed object model expression studies, only Werner's CSL provided a means of expressing refactorings to be applied to object model instances. The CSL syntax is very intuitive to read and write, but does not provide a means of extension to developers beyond the composition of existing CSL commands. The other studies reviewed in this section presented different approaches for the expression of static object models, but are not usable for the expression of refactorings that are to be applied to an object model instance.
Chapter 3

Refactoring Model

The ability of a developer to define a refactoring for application by a tool is dependent upon the existence of a common refactoring model. This refactoring model is responsible for formally specifying a set of language-independent concepts and rules that refactoring definitions may use, and consequently concepts that refactoring application tools must be able to apply appropriately. Subsequent utilization of this model requires the provision of a common means of expressing instances of it. This means of expression specifies the syntax that is used to represent model instances.

Assuming that this common refactoring model is able to represent the application of arbitrary refactorings, a tool that can fully parse expressions of this model and apply them appropriately is therefore able to apply arbitrary refactorings within its domain. This chapter describes a developed refactoring model that is able to represent the application of arbitrary object-oriented refactorings, and a formal syntax for expressing instances of this model. With these components formally specified the development of automated tool support is feasible.

3.1 Model

The role of the refactoring model described in this section is to specify a set of language-independent concepts that as a whole may be used to represent the application of arbitrary refactorings. This set of concepts must be rich enough so as to not constrain
the definition of any valid refactoring, but also relatively simple and intuitive. The model must also be unambiguous with respect to interpretation. Keeping these requirements in mind, the developed refactoring model contains sub-models describing the following three components, each of which are subsequently discussed:

- **Actors:** These are the participants (the program entities) that are involved in a refactoring. They initially provide a static view of the system entities that are relevant to a refactoring, and act as the subsequent targets of transformations.

- **Operations:** These are sets of steps that are applied to the actors in a refactoring. These steps include not only the actor transformations but also the supporting work that ensures the correct application of the transformations.

- **Execution:** These are the rules specifying how a given refactoring defined in terms of the actor and operation models is to be applied to existing software.

### 3.1.1 Actor Model

The actor model represents actors that are involved in a refactoring as typed entities. Since refactorings entail the modification of static object-oriented entities and/or their relationships with each other, the actor model is responsible for representing all such entities and relationships sufficiently and concisely in a language-independent manner. Several models for the representation of object-oriented entities exist (for example [DTS99], [Mei96] [LX93]). However none of these models is comprised of the precise set of entities and relationships that refactorings may utilize. An actor model containing precisely these entities and relationships would be guaranteed to not impose constraints
on the refactoring model’s ability to represent refactorings, since by definition all entities potentially involved in a refactoring could be represented. Such a model would also be guaranteed to be unambiguous, concise, and very intuitive given its use of well-known object-oriented concepts. Hence the actor model that is utilized by the developed refactoring model does not correspond directly to any of the reviewed object-oriented models, though it is similar to FAMIX. Its entities and their primary relationships with each other are shown in Figure 6, which utilises UML’s [uml] conventions of showing one-to-one entity relationships as labelled edges and aggregate relationships as labelled edges with diamonds.

The actor model also contains aggregate\(^1\) versions of each of these entities for the representation of actor collections. One usage of these aggregates is to depict the one-to-many relationships shown in Figure 6. It will be shown that aggregates are useful in other refactoring contexts as well. One other entity that is omitted from Figure 6 is 

\texttt{NullActor}, which can represent any unfilled actor attribute of type actor. An example usage of \texttt{NullActor} would be to represent the return type of a \texttt{Method} actor whose stated return type is \texttt{void} or some language-specific equivalent. It is not meant to represent cases where a type is not known, as is often the case in Smalltalk contexts. In such situations the type should simply be \texttt{Object}.

Each of the actor model’s entities possesses a set of internal attributes. These attributes represent the actor relationships and entity-specific properties for individual

\(^1\) The word “aggregate” is used throughout this thesis to refer to collections of entities. It is not meant to imply that the entities necessarily act as components of an object composition.
Figure 6. The primary entities and relationships in the utilized actor model
actors. Figure 7 lists the attributes for each of the respective entity types. Note that while some of the attributes are of type (aggregate) "number" and (aggregate) "string", these are not considered to be real entities. All of the attributes should be familiar to developers since they mirror standard object-oriented concepts. The "modifier" attribute appears liberally throughout the model in order to provide a means for language-specific modifiers to be attached to actors if needed, as will be elaborated on in the operation model.

Actors are always assigned unique identifiers through which they may subsequently be referenced. These identifiers communicate the role of the respective actors in the refactoring. If an actor is assigned an identifier that is already assigned to another actor then the identifier's referenced actor is changed from its original value to this new value.

3.1.2 Operation Model

Operations are logical groupings of steps that query or transform actors. A refactoring implementation is one example of an operation. Like the actor model, the operation model is subject to the same requirements of certainty and ability to represent all valid use cases. While specifying the operation model in an unambiguous manner is easily achievable, proving that it does not impose invalid constraints on operation definitions cannot be done in a direct manner since anticipating all operations that a developer could wish to define is not possible.
<table>
<thead>
<tr>
<th>Entity</th>
<th>Attribute</th>
<th>Attribute Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>parent</td>
<td>Body</td>
<td>NullActor</td>
</tr>
<tr>
<td></td>
<td>position</td>
<td>Number</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>targetAttribute</td>
<td>Attribute</td>
<td>NullActor</td>
</tr>
<tr>
<td>Attribute</td>
<td>modifiers</td>
<td>String (aggregate)</td>
<td>“private”</td>
</tr>
<tr>
<td></td>
<td>name</td>
<td>String</td>
<td>&lt;empty string&gt;</td>
</tr>
<tr>
<td></td>
<td>parent</td>
<td>Type</td>
<td>NullActor</td>
</tr>
<tr>
<td></td>
<td>type</td>
<td>Type</td>
<td>NullActor</td>
</tr>
<tr>
<td>Body</td>
<td>accesses</td>
<td>Access (aggregate)</td>
<td>&lt;empty aggregate&gt;</td>
</tr>
<tr>
<td></td>
<td>contents</td>
<td>String</td>
<td>&lt;empty string&gt;</td>
</tr>
<tr>
<td></td>
<td>invocations</td>
<td>Invocation (aggregate)</td>
<td>&lt;empty aggregate&gt;</td>
</tr>
<tr>
<td></td>
<td>modifiers</td>
<td>String (aggregate)</td>
<td>&lt;empty string&gt;</td>
</tr>
<tr>
<td></td>
<td>parent</td>
<td>Method</td>
<td>NullActor</td>
</tr>
<tr>
<td></td>
<td>position</td>
<td>number</td>
<td>0</td>
</tr>
<tr>
<td>Invocation</td>
<td>parent</td>
<td>Body</td>
<td>NullActor</td>
</tr>
<tr>
<td></td>
<td>position</td>
<td>Number</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>targetMethod</td>
<td>Method</td>
<td>NullActor</td>
</tr>
<tr>
<td>Method</td>
<td>body</td>
<td>Body</td>
<td>NullActor</td>
</tr>
<tr>
<td></td>
<td>modifiers</td>
<td>String (aggregate)</td>
<td>“public”</td>
</tr>
<tr>
<td></td>
<td>name</td>
<td>String</td>
<td>&lt;empty string&gt;</td>
</tr>
<tr>
<td></td>
<td>parameters</td>
<td>Parameter (aggregate)</td>
<td>&lt;empty aggregate&gt;</td>
</tr>
<tr>
<td></td>
<td>parent</td>
<td>Type</td>
<td>NullActor</td>
</tr>
<tr>
<td></td>
<td>returnType</td>
<td>Type</td>
<td>NullActor</td>
</tr>
<tr>
<td>Parameter</td>
<td>modifiers</td>
<td>String (aggregate)</td>
<td>&lt;empty string&gt;</td>
</tr>
<tr>
<td></td>
<td>name</td>
<td>String</td>
<td>&lt;empty string&gt;</td>
</tr>
<tr>
<td></td>
<td>parent</td>
<td>Method</td>
<td>NullActor</td>
</tr>
<tr>
<td></td>
<td>type</td>
<td>Type</td>
<td>NullActor</td>
</tr>
<tr>
<td>Type</td>
<td>attributes</td>
<td>Attribute (aggregate)</td>
<td>&lt;empty aggregate&gt;</td>
</tr>
<tr>
<td></td>
<td>methods</td>
<td>Method (aggregate)</td>
<td>&lt;empty aggregate&gt;</td>
</tr>
<tr>
<td></td>
<td>modifiers</td>
<td>String (aggregate)</td>
<td>“public”</td>
</tr>
<tr>
<td></td>
<td>name</td>
<td>String</td>
<td>&lt;empty string&gt;</td>
</tr>
<tr>
<td></td>
<td>supertypes</td>
<td>Type (aggregate)</td>
<td>&lt;empty string&gt;</td>
</tr>
</tbody>
</table>

Figure 7. The actor model’s entity attributes
The definition of an operation as a logical grouping of steps makes it analogous to a computer program. Abelson and Sussman [AS85] define the three elements that a programming language must provide in order to be usable for arbitrary computations as being:

1. Means of abstraction: The ability to name elements and manipulate them as units.
2. Means of combination: The ability to build compound elements from simpler ones.
3. Primitive expressions: The simplest entities that the language is concerned with.

The analogy between operations and a computer program allows us to prove that the operations model is not constraining with respect to expressing valid operations, and hence refactorings, if it is shown that it provides these three facilities at a minimum. The operations model contains facilities addressing these requirements, as well as additional facilities that increase its expressiveness beyond primitive expressions in ways that should be intuitive to any software developer. The full set of facilities that comprise the operation model is enumerated below and formally described:

- Operation Names
- Actor Declarations
- Actor Requirements
- Refactorings
- Assertions
- Computations
- Operation Invocations
• Assignments
• Iterations
• Conditionals
• Primitive Operations

3.1.2.1 Operation Names

Every operation must define a name for itself through which it may be referenced. The name must be unique with respect to all other operation names in the system in order to avoid referential ambiguity. Failure of an operation to define a unique name for itself results in removal of the operation and any other operations with the same name from the system's set of known operations. This requisite for defining a valid operation fulfills Abelson and Sussman's first programming element requirement.

3.1.2.2 Actor Declarations

To work with program entities, operations must have actors representing these entities. The operation model allows operations to create these actors, provided that they are able to supply the minimally required initialization information. The types of valid actor entities and their respective attributes are exhaustively described in the Actor model. Additionally, an actor's unique identifier must be defined at creation time, through which it is subsequently referenced.

3.1.2.3 Actor Requirements

Since operations utilize actors, operations necessarily require that actors fulfilling required roles be provided at execution time in order for them to perform their work. For
example, to query whether a *Type* exists in a system, an actor of entity type *Type* must be provided for which existence in the system will be tested.

The operation model meets this requirement by permitting operations to specify the actors and their respective entity types that must be accessible to the operation for it to execute. Actor requirements are specified according to their unique identifiers, and their respective entity types are identified by common type constants. If these actor requirements are not fully met at execution time then the operation fails since required actors that are to be utilized are not present.

The primary set of common entity type constants that may be used for specifying a required actor’s requisite type is the full set of actor entity types described in the actor model, excluding *NullActor*. Additionally, the operation model introduces the following entity types that may be used specifically within this context:

- **actor** and **aggregate actor**: Any actor can satisfy a requirement of type *actor*; likewise any aggregate can satisfy a requirement of type *aggregate actor*. Such requirement types are useful in situations where the type of a required actor(s) does not matter to the operation.

- **number** and **aggregate number**: Since operations may utilize actor attributes, the ability to work with numbers is vital.

- **string** and **aggregate string**: Since operations may utilize actor attributes, the ability to work with strings is vital.
3.1.2.4 Refactorings

Refactorings are represented as operations in the model, and as such are subject to the full set of requirements and operation model facilities described throughout this section. Refactorings are the primary type of defined operation, but certainly not the only one. Refactoring operations are responsible specifically for containing the steps that are required to apply a given refactoring. For the application of a refactoring operation to be considered successful, all of its contained steps must complete successfully.

3.1.2.5 Assertions

An operation must be able to check preconditions in order to verify that its application is valid given the available actors. These preconditions extend beyond checking for the presence of actors, which actor requirements already ensure, to the checking of conditions related to these actors. Operations also often check post-conditions after performing their work in order to ensure that the system is being left in a state that is consistent and as expected. The nature of post-condition checks is typically similar to that of precondition checks.

Preconditions and post-conditions are represented in the operation model as assertions for which a \textit{true} result denotes satisfaction of the required condition and a \textit{false} result denotes failure. By definition, an assertion, and hence a precondition or post-condition, will always evaluate to either \textit{true} or \textit{false}. From this point onwards preconditions and post-conditions are referred to as assertions since they use the same underlying mechanism.
Satisfaction of an assertion is determined by performing queries on the appropriate actors without modifying them in any way. This constraint on assertions of never modifying actors is hereupon referred to as its actor preservation property. If an assertion is composed of multiple queries then the overall result is the logical AND of the individual query results. Failure of an assertion results in failure of the operation.

Since assertions define steps that utilise actors to perform queries, they are considered to be another type of operation. As such they are subject to the full set of requirements and operation model facilities as discussed throughout this section.

3.1.2.6 Computations

Operations often must query for information related to actors without modifying them. An assertion provides such a query mechanism, but is constrained by its inability to indicate any result beyond either success or failure. While making such determinations is vital for defining operations, they are clearly not sufficient for performing arbitrary queries involving actors.

The operation model represents queries that can evaluate to a result of any actor or actor attribute as computations. Like assertions, computations exhibit the actor preservation property of never modifying actors. Computation results are always encapsulated in a single actor or actor attribute so that they may be referenced through a simple identifier; a computation result consisting of multiple actors/attributes is represented as an aggregate actor/attribute. There is no result value that inherently indicates failure of a computation, so this determination is left to the client.
Similar to assertions, computations perform their work by defining steps that make actor queries, and hence represent another type of operation. As such they are subject to the full set of requirements and operation model facilities as discussed throughout this section.

3.1.2.7 Operation Invocations

So far three types of operations have been identified; they are refactorings, assertions, and computations. While these types of operations are necessarily similar to each other with respect to operation model facilities, they each fulfill different essential roles that complement each other in the definition of full refactorings. Therefore a means is needed for combining operations in order to construct more complex composite operations. The ability to do so fulfills Abelson and Sussman’s second programming element requirement.

Construction of composite operations is facilitated in the operation model through the provision of operation invocations. An operation invocation is an operation step that specifies that another operation should be invoked and run to completion, barring failure, before execution of the current operation resumes. The operation to be invoked is referenced by its requisite unique name in order to guarantee unambiguous composition. Attempting to perform an invocation whose target operation name does not correspond to a known operation results in failure of the invocation step, and consequently failure of its operation.

Every invoked operation is responsible for answering a result of either success or failure to its client operation. This result indicates the operation’s success with respect to
fully completing its work, and consequently the success of the invocation as well. For assertions this result is synonymous with the assertion's satisfaction; assertion invocations can specify their success to be either directly or inversely related to this result. Failure of an invocation results in failure of the client operation.

The only required constraint on operation invocations is that refactorings are the only operations that may invoke other refactorings; this is to preserve the previously described actor preservation property of assertions and computations. Operation invocations facilitate the construction of arbitrarily complex operations respecting this constraint, and hence allow operations to perform much of their work by delegation to other operations.

3.1.2.8 Assignments

An operation that invokes another operation is responsible for ensuring the availability of actors and attributes with the required identifiers as specified by the invoked operation's actor requirements. However, actors and attributes can be acquired from a variety of different sources. As such an operation can have access to an actor or attribute that is required in order to invoke an operation, but its identifier may not match that of the invoked operation's actor requirement.

The operation model allows an actor or attribute with a given identifier to be assigned to a new identifier, thus enabling an operation to set up actor/attribute identifiers as required by an invocation's target operation. This assignment actually creates a copy of the actor/attribute, and as a result the original still remains bound to its initial identifier. The copy also results in changes that are subsequently applied to one of the
actors or attributes not affecting the other. Attempting to assign the value of an identifier that does not have an associated actor or attribute results in failure of the operation.

3.1.2.9 Iteration

The existence of aggregate actor structures (for example, a Type actor’s potential containment of multiple methods) necessitates the provision of some means of accessing the individual actors contained within them. Access to all actors involved in an operation, including those within aggregates, is clearly a requirement for enabling the definition of arbitrary refactorings.

Actors within an aggregate are accessed in the operation model using an iterator that handles all details related to indexing. The iterator repeatedly executes a set of specified steps, each time substituting a successive actor from a provided aggregate for the aggregate itself. The actors are substituted in the order in which they are contained within the aggregate. Direct indexing into aggregates is not provided as part of the operation model, though this can be achieved through the use of a computation. Attempting to iterate through an aggregate whose identifier is not known to the system results in failure of the operation.

3.1.2.10 Conditionals

An operation’s behaviour often must adapt at execution time according to specific characteristics of the involved actors. For example, an operation may only wish to invoke some other operation on Type actors that have a supertype. Without the ability to
adapt operation behaviours at execution time, operations would necessarily become too
general to facilitate representation of arbitrary refactorings.

Behavioural adaptation is provided in the operation model through the use of
conditionals that facilitate branching of execution behaviour. A conditional determines
its satisfaction based upon the evaluation of a specified assertion; the relationship
between satisfaction of the assertion and satisfaction of the conditional can be specified
to be either direct or inverse. If a conditional is satisfied then its specified steps for
execution in such cases are performed, otherwise they are not. Since assertions do not
modify actors and conditionals may specify either direct or inverse satisfaction with
respect to the tested assertion’s satisfaction, conditionals can be used to implement any
branching logic that an operation may require. Note that failure of a conditional or its
tested assertion in this context does not result in the automatic failure of the operation.
However, specifying a conditional’s associated assertion to be one whose name is not
known to the system does result in failure of the operation.

3.1.2.11 Primitive Operations

While the ability to invoke other operations provides a means for building
complex operations, it is not possible to perform an actual query or transformation on the	target system if all operations are defined solely in terms of other operations. An
anchoring mechanism is needed to work between the system being acted upon and the
defined higher-level operations. The provision of such a mechanism fulfils Abelson and
Sussman’s third programming element requirement.
The operation model meets this requirement by specifying the existence of special primitive operations whose responsibility is to act upon the target system on behalf of the higher-level operations. These primitive operations are not constrained to the types of operation model steps described here, and hence may use any means necessary for performing their work and determining their success or failure. Their sole constraint is that only refactoring primitive operations may modify the system; this is required in order to preserve the actor preservation property of assertions and computations.

Since primitive operations are necessarily tied to a specific target language as a result of their role, they perform their work in a language-specific manner. This provides a layer of abstraction between the refactoring application tool and the target environment, which allows language differences to be hidden from the defined refactorings. For example, the potential for adding a supertype to a given Type actor will be more constrained in a language like Smalltalk than in C++ since C++ allows for multiple type inheritance. As such the language-specific implementations of a hypothetical primitive assertion CanAddSuperType would each act accordingly. Similarly, primitive operations provide a means for defining queries and transformations at the abstract syntax tree level that appear to be language-independent to clients. For example, a hypothetical assertion DoesNotDefineLocalVariable could be implemented in several language-specific primitive assertions such that they would all exhibit equivalent external behaviours in spite of acting upon syntactically different languages. Primitive operations can be defined that leverage language-specific features, if desired. It is for such cases that modifier attributes are provided on all static actor model entities. However any
operations that reference language-specific primitive operations, either directly or otherwise, will consequently be tied to running in environments in which these primitive operations are available.

The ideal means for a primitive operation to perform its work is always to delegate as much of its work to the client environment as is possible. The ability to do this is of course dependent upon the willingness and ability of the target environment to perform the work. It is assumed from this point onwards that utilized primitive operations do have the means available to them to perform their required work.

3.1.3 Execution Model

Execution of an operation is performed simply by executing its contained steps in their order of appearance, as specified by the operation model. Each invoked operation executes within a new execution frame that contains its steps and actor/attribute bindings. Blocks of steps contained within steps, such as those belonging to conditionals and iterations, execute in the same execution frame as their containing step. Each actor/attribute bound within a frame may be referenced exclusively through its unique identifier. The actor/attribute identifier "RESULT" is reserved as the identifier to which computation operations should assign computed results that are to be returned to client operations.

The execution frames of successively invoked operations are dynamically scoped, so a frame has read access to actor/attribute bindings in its own frame and all ancestor frames in their reverse order of invocation. Consequently an actor/attribute lookup will
always evaluate to the first value found with the required identifier. However assignments of actor bindings are always performed in the immediate execution frame, so a binding established in a given frame cannot be reassigned as a side effect of invoking an operation. Failure to find an actor/attribute corresponding to a referenced identifier results in failure of the operation.

Failure of an operation causes execution of its steps to cease and reverts the thread of execution to the client that invoked it. Since failure of an operation typically results in failure of the client’s invocation step, this execution thread unwinding will often lead to the root refactoring operation eventually failing. In such cases all changes that have been made as a result of the initial invocation of the root refactoring must be undone so that the state of the system is left unchanged. However this fate cannot be assumed for all failed operations since some invocations, such as assertion invocations, are able to specify their success to vary inversely to that of their invoked operation.

3.2 Expression of Refactorings

In order to express instances of the refactoring model such that they are applicable by a tool, a standard expression notation is required. This notation must be able to represent any instance of the model in an unambiguous manner. Additionally, since these refactorings are intended to be sharable across various tools often acting upon different target languages, the notation must be tool-independent and language-independent in nature.
XML can be used as a means for concretely expressing refactorings. XML is a simple markup language that is widely used for the plaintext representation of both models and model instances. Models are specified in XML in the form of Document Type Definitions (DTD), and instances of models are defined using markup corresponding to the model's definition. This separation makes validation of model instances with respect to compliance with a given model very straightforward. The plaintext nature of XML documents makes them easy to transport and manipulate by users. These stated characteristics of XML make it an ideal means for concretely expressing refactorings.

This section examines the DTD segments that have been specified for representing the various aspects of the presented refactoring model. This DTD determines the extent to which a given refactoring expression can be validated for compliance with the refactoring model using purely syntactic mechanisms. The complete DTD is shown in Appendix A.

3.2.1 Expression of Actor Declarations

As was described in the Actor Model section, every actor in a refactoring has an entity type from the set of predefined entity types, a unique identifier, and a set of attributes that is appropriate for its entity type as shown in Figure 7. Given this common template of what constitutes an actor, the refactoring DTD specifies the notations for all actor declarations in a similar manner. Actor declarations are represented in the DTD as XML elements, since the purpose of elements is to represent "real" structures that can
stand on their own [StL00]. The tag for an actor declaration is the actor’s entity type. An actor’s unique identifier is specified to be an XML attribute since the identifier is considered to be an attribute of the actor to which it is attached. Actor attributes are also represented as XML attributes since they are entity-specific. For attributes that are specified to be optional, denoted by the XML #IMPLIED symbol, the default attribute value is assumed whenever an overriding value is not explicitly provided. Figure 8 shows the DTD declarations for the primary actor types.

While string and number are not considered to be actor types, their roles as actor attribute types necessitates the provision of a means for defining and manipulating instances of them. Each of these entities has a unique identifier attribute for future reference, and an attribute representing its value. They are specified in a similar manner to actor entities, as is shown in Figure 9.

The aggregate entities are specified in a similar manner to the singular entities. However, since aggregates simply act as actor containers, their only attributes are their respective unique identifiers. Figure 10 shows the declarations for all of the aggregate entity types.

3.2.2 Expression of the Operation Model

Similar to actor declarations, steps in an operation are specified as XML elements. Step-specific information is contained in associated element attributes. Representing steps as elements best depicts their self-contained nature and makes them embeddable within other elements as appropriate. The operation model entities for which steps may
<!ELEMENT access EMPTY>
<!ATTLIST access
  id CDATA #REQUIRED
  parentId CDATA #REQUIRED
  positionId CDATA #IMPLIED
  targetAttributeId CDATA #REQUIRED>

<!ELEMENT attribute EMPTY>
<!ATTLIST attribute
  id CDATA #REQUIRED
  modifiersId CDATA #IMPLIED
  nameId CDATA #REQUIRED
  parentId CDATA #REQUIRED
  typeId CDATA #REQUIRED>

<!ELEMENT body EMPTY>
<!ATTLIST body
  id CDATA #REQUIRED
  accessesId CDATA #IMPLIED
  contentsId CDATA #IMPLIED
  invocationsId CDATA #IMPLIED
  modifiersId CDATA #IMPLIED
  parentId CDATA #REQUIRED
  positionId CDATA #IMPLIED>

<!ELEMENT invocation EMPTY>
<!ATTLIST invocation
  id CDATA #REQUIRED
  parentId CDATA #REQUIRED
  positionId CDATA #IMPLIED
  targetMethodId CDATA #REQUIRED>

<!ELEMENT method EMPTY>
<!ATTLIST method
  id CDATA #REQUIRED
  bodyId CDATA #IMPLIED
  modifiersId CDATA #IMPLIED
  nameId CDATA #REQUIRED
  parametersId CDATA #IMPLIED
  parentId CDATA #REQUIRED
  returnTypeId CDATA #IMPLIED>
Figure 8. DTD for the primary actor entity types
<!ELEMENT number EMPTY>
<!ATTLIST number
  id CDATA #REQUIRED
  value CDATA #REQUIRED>

<!ELEMENT string EMPTY>
<!ATTLIST string
  id CDATA #REQUIRED
  content CDATA #REQUIRED>

Figure 9. DTD for the Number and String entity types
<!ELEMENT access.aggregate EMPTY>
<!ATTLIST access.aggregate id CDATA #REQUIRED>

<!ELEMENT attribute.aggregate EMPTY>
<!ATTLIST attribute.aggregate id CDATA #REQUIRED>

<!ELEMENT body.aggregate EMPTY>
<!ATTLIST body.aggregate id CDATA #REQUIRED>

<!ELEMENT invocation.aggregate EMPTY>
<!ATTLIST invocation.aggregate id CDATA #REQUIRED>

<!ELEMENT method.aggregate EMPTY>
<!ATTLIST method.aggregate id CDATA #REQUIRED>

<!ELEMENT parameter.aggregate EMPTY>
<!ATTLIST parameter.aggregate id CDATA #REQUIRED>

<!ELEMENT type.aggregate EMPTY>
<!ATTLIST type.aggregate id CDATA #REQUIRED>

<!ELEMENT number.aggregate EMPTY>
<!ATTLIST number.aggregate id CDATA #REQUIRED>

<!ELEMENT string.aggregate EMPTY>
<!ATTLIST string.aggregate id CDATA #REQUIRED>

Figure 10. DTD for all aggregate entity types
be embedded are conditional steps, iteration steps and operations (refactorings, assertions and computations). The three operation types are each represented as XML elements with an associated attribute for containing the required unique name. The operation model concepts → XML mappings are listed below, with backward references to refactoring model sections shown in parenthesis:

- actor requirement (3.1.2.3) → element “requires”
  - the required identifier of the actor → attribute “id”
  - the required type of the actor → attribute “entity”
- refactoring declaration (3.1.2.4) → element “refactoring”
  - the name of the refactoring (3.1.2.1) → attribute “name”
- assertion declaration (3.1.2.5) → element “assertion”
  - the name of the assertion (3.1.2.1) → attribute “name”
- computation declaration (3.1.2.6) → element “computation”
  - the name of the computation (3.1.2.1) → attribute “name”
- refactoring invocation (3.1.2.7) → element “refactor”
  - the name of the refactoring to invoke → attribute “name”
- assertion invocation (3.1.2.7) → element “assert”
  - the name of the assertion to invoke → attribute “name”
  - the modifier to apply to the result in order to determine satisfaction → attribute “modifier” (default value is none)
- computation invocation (3.1.2.7) → element “compute”
  - the name of the computation to invoke → attribute “name”
• the actor id to put the computation’s result into \( \rightarrow \) attribute “resultId”

• assignment (3.1.2.8) \( \rightarrow \) element “assign”
  ○ the new actor/attribute identifier \( \rightarrow \) attribute “id”
  ○ the source actor/attribute identifier \( \rightarrow \) attribute “valueId”

• iteration declaration (3.1.2.9) \( \rightarrow \) element “iterate”
  ○ the aggregate to iterate over \( \rightarrow \) attribute “aggregateId”

• conditional declaration (3.1.2.10) \( \rightarrow \) element “cond”
  ○ the name of the assertion to invoke in order to determine the condition’s satisfaction \( \rightarrow \) attribute “name”
  ○ the modifier to apply to the assertion result in order to determine satisfaction of the conditional \( \rightarrow \) attribute “modifier” (default value is none)

Actor declarations (3.1.2.2) were already described in the previous section. The only operation model entity that does not have a corresponding XML representation is the primitive operation (3.1.2.11), since by definition these are target-specific. Figure 11 shows the DTD declarations for the entities in the operation model.

The only shortcoming of the presented operation model expression is its failure to fully capture the actor preservation property of assertions and computations. Specifically, the sub element lists for elements cond and iterate cannot be specified to only allow inclusion of the refactor sub element when the operation element is a refactoring. This is a constraint of DTDs that cannot be overcome. Inclusion of the refactor sub element for these two elements is necessary so that refactorings are not constrained from invoking other refactorings from within conditionals or iterations. As a
<!ENTITY % allActorTypes "access | access.aggregate | attribute |
 attribute.aggregate | body | body.aggregate | invocation |
 invocation.aggregate | method | method.aggregate | parameter |
 parameter.aggregate | type | type.aggregate | actor ">
<!ENTITY % allTypes "%allActorTypes; | number | number.aggregate |
 string | string.aggregate | actor.aggregate ">

<!ELEMENT requires EMPTY>
<!ATTLIST requires
  id CDATA #REQUIRED
  entity (%allTypes;) #REQUIRED
  >

<!ELEMENT assert EMPTY>
<!ATTLIST assert
  name CDATA #REQUIRED
  modifier (not) #IMPLIED
  >

<!ELEMENT compute EMPTY>
<!ATTLIST compute
  name CDATA #REQUIRED
  resultId CDATA #REQUIRED
  >

<!ELEMENT refactor EMPTY>
<!ATTLIST refactor
  name CDATA #REQUIRED
  >

<!ELEMENT assign EMPTY>
<!ATTLIST assign
  id CDATA #REQUIRED
  valueId CDATA #REQUIRED
  >

<!ELEMENT iterate (refactor | assert | compute | iterate | assign | cond |
  %allTypes;)*>
<!ATTLIST iterate aggregateld CDATA #REQUIRED>

<!ELEMENT cond (refactor | assert | compute | iterate | assign | cond |
  %allTypes;)*>
<!ATTLIST cond
  name CDATA #REQUIRED
  modifier (not) #IMPLIED
>
</!ELEMENT refactoring (requires | refactor | assert | compute | iterate | assign |
  cond | %allTypes;)*>
<!ATTLIST refactoring
  name CDATA #REQUIRED
>
</!ELEMENT assertion (requires | assert | compute | iterate | assign | cond |
  %allTypes;)*>
<!ATTLIST assertion
  name CDATA #REQUIRED
>
</!ELEMENT computation (requires | assert | compute | iterate | assign | cond |
  %allTypes;)*>
<!ATTLIST computation
  name CDATA #REQUIRED
>

Figure 11. DTD for all operation model entities
result, assertions and computations whose steps invoke refactorings from within conditionals or iterations will be evaluated to be valid by XML syntax validators according to the specified DTD, but must fail at execution time as a result of non-compliance with the refactoring model. This shortcoming is the only aspect for which an operation's compliance with the refactoring model cannot be verified by a standard XML validator.

3.3 Summary

This chapter has formally specified and presented the two components that facilitate the implementation of tools that can be used to automatically apply any object-oriented refactoring. The first presented component was a common refactoring model, which provides a set of language-independent concepts for defining a refactoring in terms of. A refactoring that is defined strictly in terms of these concepts will be "understood" by any client that is fully aware of the concepts and their prescribed application rules. Language-specific behaviours and actor characteristics are isolated within primitive operations and actor modifier attributes, respectively. The second presented component was syntax for expressing instances of this model. This syntax provides a standard means for exchanging refactoring model instances, and hence enables a refactoring definer to communicate any refactoring to tools that are able to parse it. The tools are subsequently in a position to apply the refactoring in an automated manner as prescribed by the refactoring model. One such tool has been developed as part of this research, and is discussed in the next chapter.
Chapter 4

Application of Refactoring Model

The refactoring model and expression syntax presented in the previous chapter provide the components that are needed for communicating a refactoring application to an automated tool. This chapter demonstrates the range of capabilities of these components in order to show the model's ability for defining arbitrary refactorings. Two complete refactorings are presented that have been defined in terms of the presented refactoring model. Also shown are the respective expressions of these refactorings that use the prescribed expression syntax. Finally, a tool is described that has been developed for the purpose of applying such refactorings in a common integrated development environment.

4.1 Example Refactorings

This section demonstrates how the refactoring model and associated expression syntax are used to define two well-known refactorings. The refactorings that are shown are AddParameter [FBBO99] and ExtractMethod [FBBO99]. These are commonly used refactorings that any refactoring tool should be able to apply. They were chosen because their respective application characteristics are opposite to each other in terms of the level of required program code manipulations. This variance allows for the demonstration of a wide range of the refactoring model's capabilities, and implies the model's ability to represent other refactorings as well whose application characteristics fall between these
extreme cases. Additionally, the broad refactoring model use that is made by these two refactorings allows for the demonstration of much of the specified expression syntax. Each refactoring’s description progresses from a general descriptive passage to refactoring model concepts to concrete expression in order to illustrate the steps that are typically taken to define any refactoring for application by the automated tool.

4.1.1 Example Refactoring: AddParameter

The goal of the AddParameter refactoring is to add a parameter to an existing method without affecting program behaviour. This is considered to be a high-level refactoring because it deals primarily with the manipulation and querying of higher-level object-oriented structures such as methods and method parameters. This refactoring demonstrates the refactoring model’s abilities to work with these higher-level entities. The typical motivation for performing this refactoring is to allow clients of a method to provide more information to it at invocation time. The refactoring is concerned specifically with the proper addition of the new parameter that will facilitate this; the developer is subsequently tasked with modifying the target method to utilise the parameter as needed.

While the adding of the new parameter to the method is quite straightforward, doing so in isolation can potentially introduce negative side effects. These side effects, and strategies for avoiding them, are:

- Loss of polymorphism in the type hierarchy: If the method was initially polymorphic with other methods in its type hierarchy then adding a parameter to
the method will change this relationship. This is a very bad problem to introduce because it often will not result in a compile error. To avoid this problem all methods in the type hierarchy that are polymorphic with respect to the method receiving the new parameter should also have the new parameter added to them. This will maintain the polymorphic relationship amongst these methods.

- Invalidation of client references to the method: Adding a parameter to a method changes its signature. As a result, any client that invoked the method will no longer be valid, assuming that all polymorphic methods in the type hierarchy received new parameters as well. The fix for this problem is to add an empty argument to all clients of the method. This will re-establish the validity of the client references by making them reference the original method through its new signature.

Application of the AddParameter refactoring is guaranteed to be behaviour preserving if the two side effect avoidance strategies above are fully applied, assuming that they do not result in conflicts either amongst argument names within a method or amongst method signatures in the type hierarchy. The following section details how to represent all of this information in terms of the refactoring model.

4.1.1.1 Applicability of the Refactoring Model

The AddParameter refactoring utilizes the following primary actors:

- The parameter that is to be added to the method.

- The method that is to receive the new parameter.
• Methods in the type hierarchy that are polymorphic with respect to the method that is receiving the new parameter.

• Methods that invoke either the method that is receiving the new parameter or polymorphically equivalent methods in the type hierarchy.

The operation model steps in this refactoring are described next. Note that enumerated steps in this section use a continuous numbering scheme in order to facilitate backward references by its expression shown in the next section.

The initial actors that this refactoring utilizes are:

1. The method that is to receive the new parameter.

2. The parameter that is to be added to this method.

The first actor represents an existing target entity, and is therefore a required actor that should be supplied by the refactoring client. Since the second actor is a parameter that does not yet exist, the refactoring should use a computation to query the user for its name and (optionally) type. Once these actors have been determined then two actor preconditions must be satisfied in order for application of the refactoring to be valid:

3. (Precondition) Methods in the type hierarchy that are polymorphic with the method that is to receive the parameter cannot have another parameter with the same name. This is because a method cannot have more than one parameter with a given name. To verify this:

   a. Use a computation to determine this set of polymorphic methods. The assumption is made for this example that required operations are available for invocation, either in the form of developer-defined operations or system
primitive operations. The implementations of such supporting operations are omitted from this example solely for brevity. A developer would implement a computation such as the one required by this step by using a similar approach to that which is used to develop this refactoring.

b. Use an iterator to go through these methods and assert that each does not have a parameter with the same name as the parameter to be added.

4. (Precondition) Adding the supplied parameter to the method cannot result in a signature conflict with any other method in the type hierarchy. This is to ensure polymorphism relationships that did not previously exist are not introduced. To verify this:

   a. Use a computation to determine what the method's new signature will be (it will be its old signature with the new parameter added). Create a method actor with this signature.

   b. Use a computation to determine the set of methods in the type hierarchy that have a matching signature. Assert that no such methods were found.

If these preconditions are satisfied then the refactoring should be applied as was described at the beginning of the section. The application steps are:

5. Update all methods in the type hierarchy that are polymorphic with the method that is to receive the new parameter. To do this:

   a. Use a computation to determine this set of polymorphic methods.

   b. Use an iterator to repeatedly invoke a primitive refactoring that adds the new parameter to each of these methods.
6. Update all invokers of these modified methods. To do this:
   
a. Use a computation to determine this set of method invokers.

   b. Use an iterator to repeatedly invoke a primitive refactoring that adds an empty argument to each of these invokers.

4.1.1.2 Expression of this Refactoring

   The XML expression of this refactoring appears below with explanatory comments embedded within it. Figure 12 shows the refactoring itself, and its precondition is shown in Figure 13. Some of the comments contained within these figures have step references added to them in order to provide a mapping from the comment to the previously described set of steps. For example, "(5)" refers to step 5.

4.1.2 Example Refactoring: ExtractMethod

   The goal of ExtractMethod is to extract a given block of contiguous code from a method's body into a new method in the same class. Modifying the original method to invoke this new method in the spot from which the block of code was originally extracted then restores behaviour preservation. Unlike the AddParameter refactoring, the majority of this refactoring's program analysis and manipulation occurs at the method body level, making this a low-level refactoring. This refactoring demonstrates the refactoring model's ability to work with programs at a low level that is comparable to that of abstract syntax trees. As a result of the increased complexity that is inherently associated with working with programs at this level, the ExtractMethod refactoring is considered to be an excellent refactoring for demonstrating the capabilities of refactoring tools in general.
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE refactoring SYSTEM "http://www.scs.carleton.ca/~ggayed/refactoring.dtd">

<refactoring name="AddParameterToMethod">

<!-- List the actors that must be supplied (1) -->
<requires id="method" entity="method"/>

<!-- Query for the new Parameter's info, and create an actor accordingly (2) -->
<actor id="newParameterType"/>
<cond name="IsLanguageTyped">
  <string
    id="message"
    content="What is the new parameter's type?"/>
  <compute name="QueryUserForType" resultId="newParameterType"/>
</cond>
<compute name="QueryUser" resultId="newParameterName"/>
<parameter
  id="parameter"
  typeId="newParameterType"
  nameId="newParameterName"
  parentID="method"/>

<!-- Precondition (3, 4) -->
<assert name="CanAddParameterToMethod"/>

<!-- Perform the refactoring's steps -->
<assign id="originalMethod" valueId = "method"/>

<!-- Update polymorphic methods in the type hierarchy (5) -->

<!-- Determine polymorphic methods in type hierarchy (5a) -->
<compute
  name="MatchMethodSignatureInHierarchy"
  resultId="methodMatches"/>

<!-- Iterate through each of these methods (5b) -->
<iterate aggregateId="methodMatches"/>
<!-- Add the new parameter to the current method (5b) -->
<refactor name="prim-AddParameterToMethod"/>
</iterate>

<!-- Update invokers of the original methods (6) -->
<assign id="method" valueId="originalMethod"/>

<!-- Determine all invokers of the original method (6a) -->
<compute name="DetermineMethodReferences" resultId="invokers"/>

<!-- Iterate through each of these invokers (6b) -->
<iterate aggregateId="invokers">
  <assign id="invocation" valueId="invokers"/>
  <!-- Add a null argument to the current invoker (6b) -->
  <refactor name="prim-AddNullArgumentToInvocation"/>
</iterate>
</refactoring>

Figure 12. Expression of the AddParameter refactoring
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE assertion SYSTEM
  "http://www.scs.carleton.ca/~ggayed/refactoring.dtd">

<assertion name="CanAddParameterToMethod">

<!-- List the actors that must be supplied (1, 2) -->
<requires id="parameter" entity="parameter"/>
<requires id="method" entity="method"/>

<!-- Perform the assertion's steps -->
<assign id="originalMethod" valueId="method"/>

<!-- Check for parameter name collisions in hierarchy (3) -->

<!-- Determine the param name to check for in these polymorphic methods -->
<compute name="GetParameterName" resultId="name"/>

<!-- Determine polymorphic methods in the Type hierarchy (3a) -->
<assign id="actor" valueId="method"/>
<compute name="GetParent" resultId="methodType"/>
<compute
  name="MatchMethodSignatureInHierarchy"
  resultId="methodMatches"/>

<!-- Iterate through these polymorphic methods (3b) -->
<iterate aggregateId="methodMatches">
  <assign id="method" valueId="methodMatches"/>
  <!-- Assert there are no same-name parameters in current method (3b) -->
  <compute
    name="ExtractMethodParametersWithName"
    resultId="nameMatches"/>
  <assign id="aggregate" valueId="nameMatches"/>
  <assert name="AggregateIsEmpty"/>
</iterate>

<!-- Check for potential signature collisions in hierarchy (4) -->

<!-- Determine the new method signature with the parameter added (4a) -->
<assign id="method" valueId="originalMethod"/>
<compute name="GetMethodParameters" resultId="parameters"/>
<assign id="aggregate" valueId="parameters"/>
<assign id="actor" valueId="parameter"/>
<compute name="AddActorToAggregate" resultId="parameters"/>

<!-- Create a Type actor with this new signature (4a) -->
<compute name="GetMethodName" resultId="name"/>
<method id="newMethod"
  nameId="name"
  parametersId="parameters"
  parentId="methodType"/>

<!-- Assert no signature collisions with pre-existing methods in hierarchy (4b) -->
<assign id="method" valueId="newMethod"/>
<compute
  name="MatchMethodSignatureInHierarchy"
  resultId="methodMatches"/>
<assign id="aggregate" valueId="methodMatches"/>
<assert name="AggregateIsEmpty"/>

</assertion>

Figure 13. Expression of the CanAddParameterToMethod assertion
The typical motivation for performing this refactoring is to break a large method into smaller methods that each performs more concentrated units of work. This can make methods easier to understand as a result of their simplified nature. Another motive for using this refactoring is to extract code that is duplicated in multiple methods into one method that can subsequently be shared.

Like the demonstration of the AddParameter refactoring, this demonstration will be defined such that it mirrors the definition of the refactoring by Fowler. This clarification is made here because Fowler makes some simplifying assumptions that are not always valid. Specifically, Fowler ignores complications that can accompany the extraction of code blocks that can throw exceptions or return. He does this in order to handle the typical use case of the refactoring without introducing excess complication. While following this lead will simplify the example that is presented here, it will not reduce the scope of the refactoring model and associated expression syntax that is utilized. Rather, it will reduce redundancies in the use of these facilities.

Unlike the AddParameter refactoring, most of the work associated with ExtractMethod involves modifying the participant actors rather than making a small change and then resolving the resulting side effects. The most complicated aspect of applying this refactoring is the management of local variables, which often must be passed between the original method and extracted method to ensure that the validity of all accesses and modifications is preserved. These local variables are passed as arguments and return values in the invocation of the new method from the original method. This
approach is clean and straightforward with respect to keeping the required changes local, but imposes some constraints on the contexts in which this refactoring can be used.

The criteria for determining whether a given block of code can be extracted from a method are listed below, followed by a list of the steps for applying the refactoring. Like the previous example, the steps in this example are enumerated using a continuous numbering scheme in order to facilitate backward reference by subsequent sections.

1. Split the original method body into three exclusive, contiguous blocks of code.

   These blocks are:

   a. Extraction body: This is the block of code that is to be extracted into a new method.

   b. Upper body: This is the block of code that spans from the beginning of the method body to the beginning of the extraction body.

   c. Lower body: This is the block of code that spans from the end of the extraction body to the end of the method body.

2. Ensure that the extraction body assigns to no more than one local variable that is also accessed in the lower body. This constraint applies because only one value can be returned to the original method through the new method invocation.

3. Ensure that the extraction body will compile in isolation. This test can make the assumption that local variables utilized by the extraction body but defined by the upper body will be available in the form of method parameters.

4. Ensure that the original method body will compile without the extraction body. This test can make the assumption that a local variable that is defined in the
extraction body but also used in the lower body will be defined in the original body.

Once the stated preconditions have been satisfied then the steps involved in applying the refactoring are as follows:

5. Define the new method. It will be comprised of:
   a. Return type: This will be the type of the local variable, if any, that is assigned to in the extraction body and accessed in the lower body.
   b. Name: This will be a name of the user’s choice.
   c. Parameter types: These will be the types of all local variables, if any, that are defined in the upper body and used in the extraction body.
   d. Body: This will be the content of the extraction body. If this body assigns to a local variable that is accessed in the lower body then add a statement to this body to return the value of this variable to the invoker.

6. Ensure that the signature of this determined method does not collide with any other method in the type hierarchy. This condition ensures that polymorphism is not introduced unintentionally.

7. Create the new method.

8. Define the invocation body that will invoke the new method from the original method. Ensure that it passes the arguments in the same order in which the new method is expecting them.

9. If the new method returns a local variable that is accessed in the lower body then assign the return value of the invocation body to this variable.
10. If this returned local variable is not defined in the upper body then add a declaration for it that precedes the invocation body.

11. Set the body of the original method to be the concatenation of the upper body, invocation body and lower body.

4.1.2.1 Applicability of the Refactoring Model

The ExtractMethod refactoring utilizes the following primary actors:

- The Body of the original method.
- The Body of the original method that is to be extracted.
- The upper and lower Body blocks of the original method Body.
- The new Method that is created, and its contained Parameters and Body.

The operation model steps in this refactoring are described next. Note that enumerated steps in this section correspond with the steps in the previous section accordingly. The actors that are initially used in this refactoring are:

- The original method Body that is to be extracted from (the extraction body).
- The subset of the original method Body that is to be extracted.

The method body to be extracted clearly must be provided by the refactoring client. From this the full body of the method can be computed.

Once these actors are determined then the previously described criteria for determining whether the provided Body can be extracted must be verified. The refactoring model steps for verifying these criteria are:
1. Determine the original method’s upper body, extraction body and lower body.
   a. The extraction body is already provided.
   b. Use a computation to determine the portion of the original method Body that spans from position 0 to the position of the extraction body. Create a new Body actor, called upper body, with this content.
   c. Use a computation to determine the portion of the original method Body that spans from the end of the extraction body to the end of the original method body. Create a new Body actor, called lower body, with this content.

2. Ensure that no more than one local variable is modified in the extraction body and accessed in the lower body.
   a. Use a computation to determine the set of local variables that are used in the extraction body.
   b. Use an iterator to evaluate conditionals for each of these variables, to determine which ones are both modified in the extraction body and accessed in the lower body. Use a computation to increment a counter for each of the variables that satisfy both of these conditionals, and store such variables for future reference.
   c. Assert that the counter’s value is not greater than one.

3. Ensure that the extraction body will compile in isolation.
   a. Use a computation to determine the set of local variables that are used in the extraction body.
b. Use an iterator to evaluate a conditional for each of these variables, to determine which ones are defined in the extraction body. Use a computation to temporarily add a variable declaration to the extraction body for each of the variables that does not satisfy the conditional.

c. Assert that this computed Body still compiles without problem.

4. Ensure that the original method body will compile without the extraction body.
   a. Evaluate a conditional to determine whether step 2 detected a local variable that was assigned to in the extraction body and accessed in the lower body.
   b. If this conditional is satisfied then invoke a conditional that determines whether this variable is defined in the upper body.
   c. If this conditional is not satisfied then use a computation to temporarily add a declaration of this variable to the upper body.
   d. Use a computation to concatenate this computed upper body and lower body.

      Create a Body actor with this content

   e. Assert that this body will compile without problem.

Satisfaction of these conditions indicates that extraction of the specified Body is valid. The refactoring model steps for applying the refactoring to the provided actors are:

5. Define the new Method.
   a. Return type:
      i. Evaluate a conditional to determine whether step 2 detected a local variable that was assigned to in the extraction body and accessed in the lower body.
ii. If this conditional is satisfied then use a computation to determine this variable’s type. Set the return type to this type.

b. Name: Use a computation to query the user for the new method’s name.

c. Parameters:

i. Use a computation to determine the set of local variables that are used in the extraction body.

ii. Use an iterator to evaluate a conditional for each of these variables, to determine which ones are defined in the extraction body.

iii. Create a new Parameter actor for each of the variables that does not satisfy this conditional. Set each parameter’s name and type to that of the respective local variable. Use a computation to add this parameter to an aggregate Parameter actor that will define the new method’s set of parameters.

d. Body:

i. Create a Body actor with the contents of extraction body.

ii. Evaluate a conditional to determine whether step 2 detected a local variable that was assigned to in the extraction body and accessed in the lower body.

iii. If this conditional is satisfied then use a computation to add a statement to the end of the new body that returns this variable.

6. Assert that the signature of the new method does not collide with any other methods in the type hierarchy.
7. Invoke a refactoring to concretely create this new method.

8. Define the invocation body.
   a. Create a Body actor to represent the invocation of the new method from the original method. To compose this invocation body:
      i. Invocation arguments: Use an iterator to create new Body actors with the names of each of the previously computed parameters of the new method. This is done because these parameters represent arguments that are to be passed from the original method to the new method. Use a computation to add each of these bodies to an aggregate body that will define the invocation’s set of arguments.
      ii. Method name: This is the name of the new method.
   b. Use a computation to add an invocation statement with this name and set of parameters to the new invocation body.

9. Assign the result of the invocation to a local variable if it is accessed in the lower body.
   a. Evaluate a conditional to determine whether step 2 detected a local variable that was assigned to in the extraction body and accessed in the lower body.
   b. If this conditional is satisfied then use a computation to add an assignment of the method invocation result to a local variable with this name.

10. Add a local variable declaration for the returned value of the invocation to the invocation body if needed.
a. If an assignment was added in step 9 then evaluate a conditional to determine
   if this variable is declared in the upper body.

b. If this conditional is not satisfied then use a computation to add a declaration
   of this variable to the invocation body, preceding the invocation statement.

11. Update the body of the original method.

   a. Use computations to concatenate the upper body, invocation body and lower
      body. Create a new Body actor with this content.

   b. Use a refactoring to set the original method’s body to this new body.

4.1.2.2 Expression of this Refactoring

   The XML expression of this refactoring and its preconditions is shown in Figures
   14 and 15. Similar to the previous example expression, these figures contain explanatory
   comments that often reference steps from the previous section by step number.

4.2 Refactoring Tool

   A tool has been developed for applying defined refactorings to existing programs
   in the Eclipse IDE. The tool is implemented as an extension of Eclipse’s Java UI plugin,
   and as such its default set of primitive operations utilize Eclipse’s exposed JDT (Java
   Development Toolkit) interface to act upon its contained Java entities. However,
   different sets of primitive operations may be defined for use, thereby allowing the tool to
   act upon programs written in languages other than Java, and even potentially within other
   environments if desired.
<xml version="1.0" encoding="UTF-8"/>
<!DOCTYPE refactoring SYSTEM
    "http://www.scs.carleton.ca/~ggayed/refactoring.dtd">

<refactoring name="ExtractMethod">
  <!-- List the actors that must be supplied -->
  <requires id="extractionBody" entity="body"/>

  <!-- Determine full method body -->
  <assign id="actor" valueId="extractionBody"/>
  <compute name="GetParent" resultId="method"/>
  <compute name="GetMethodBody" resultId="fullBody"/>

  <!-- Preconditions (2-4) -->
  <assert name="CanExtractBlock"/>

  <!-- Split original body into upperBody, extractionBody and lowerBody (1) -->
  <number id="start" value="0"/>
  <!-- upperBody (1b) -->
  <assign id="body" valueId="extractionBody"/>
  <compute name="GetBodyPosition" resultId="end"/>
  <assign id="body" valueId="fullBody"/>
  <compute name="ExtractBodyRange" resultId="upperBody"/>
  <!-- lowerBody (1c) -->
  <assign id="number1" valueId="end"/>
  <assign id="body" valueId="extractionBody"/>
  <compute name="GetBodyContent" resultId="string"/>
  <compute name="DetermineStringLength" resultId="bodyLength"/>
  <assign id="number2" valueId="bodyLength"/>
  <compute name="AddNumbers" resultId="lowerBodyStart"/>
  <assign id="start" valueId="lowerBodyStart"/>
  <assign id="body" valueId="fullBody"/>
  <compute name="GetBodyContent" resultId="string"/>
  <compute name="DetermineStringLength" resultId="end"/>
  <compute name="ExtractBodyRange" resultId="lowerBody"/>

  <!-- Define the new Method (5) -->
  <number id="position" value="0"/>

  <!-- Determine the new method's return type (5a) -->
<!-- Determine the local variable, if any, that is modified in -->
<-- extractionBody and accessed in lowerBody (5a-i)-->
<actor id="modifiedLocal"/>
<compute
  name="DetermineAccessedLocalVariables"
  resultId="localVariables"/>
<iterate aggregateId="localVariables">
  <assign id="body" valueId="extractionBody"/>
  <assign id="variable" valueId="localVariables"/>
  <cond name="BodyAssignsLocalVariable">
    <assign id="body" valueId="lowerBody"/>
    <cond name="BodyAccessesLocalVariable">
      <!-- Such a variable was found -->
      <assign id="modifiedLocal" valueId="variable"/>
    </cond>
  </cond>
</iterate>

<-- The new method return type will be the type of this variable, if any (5a-ii)-->
<actor id="returnVarType"/>
<assign id="actor" valueId="modifiedLocal"/>
<cond name="ActorIsNull" modifier="not">
  <assign id="variable" valueId="modifiedLocal"/>
  <assign id="body" valueId="fullBody"/>
  <compute name="DetermineVariableType" resultId="returnVarType"/>
</cond>

<-- Query for the name of the new method (5b) -->
<string
  id="message"
  content="What is the new method's name?"/>
<compute name="QueryUser" resultId="newMethodName"/>

<-- Determine the new method's parameters (5c) -->
<assign id="actor" valueId="extractionBody"/>
<compute name="GetParent" resultId="parent"/>
<parameter aggregate id="parameters"/>
<assign id="body" valueId="extractionBody"/>
<compute
  name="DetermineAccessedLocalVariables"
  resultId="localVariables"/>
<!-- Iterate through all localVariables accessed by extractionBody, look for -->
<!-- those not defined in extractionBody (so defined in upperBody) (5c-ii) -->
<assign id="variable" valueId="localVariables"/>
<iterate aggregateId="localVariables">
  <assign id="body" valueId="extractionBody"/>
  <assign id="variable" valueId="localVariables"/>
  <cond name="BodyDefinesLocalVariable" modifier="not">
    <!-- Such a variable was found. Create a parameter so the -->
    <!-- original method can provide this variable's value to the -->
    <!-- new method as an invocation argument (5c-iii) -->
    <assign id="body" valueId="upperBody"/>
    <compute name="DetermineVariableType" resultId="variableType"/>
    <assign id="body" valueId="variable"/>
    <compute name="GetBodyContent" resultId="variableName"/>
    <parameter id="parameter"
      nameId="variableName"
      parentId="parent"
      typeId="variableType"/>
    <assign id="actor" valueId="parameter"/>
    <assign id="aggregate" valueId="parameters"/>
    <compute name="AddActorToAggregate" resultId="parameters"/>
  </cond>
</iterate>

<!-- Create the new method's Body, set its contents to extractionBody (5d-i) -->
<assign id="body" valueId="extractionBody"/>
<compute name="GetBodyContent" resultId="methodContents"/>
<body id="newBody"
  contentsId="methodContents"
  parentId="parent"/>

<!-- Add trailing "return..." to new method's Body if needed (5d-ii) -->
<assign id="actor" valueId="modifiedLocal"/>
<cond name="ActorIsNull" modifier="not">
  <assign id="body" valueId="newBody"/>
  <assign id="variable" valueId="modifiedLocal"/>
  <compute name="AddReturnDeclaration" resultId="newBody"/>
</cond>

<!-- Create the new Method -->
<assign id="actor" valueId="parent"/>
<compute name="GetParent" resultId="class"/>
<method
   id="method"
   bodyId="newBody"
   nameId="newMethodName"
   parentId="class"
   parametersId="parameters"
   returnTypeId="returnVarType"/>

<!-- Ensure that this new method will not collide with another method in the-->
<!-- type hierarchy (6) -->
<compute
   name="MatchMethodSignatureInHierarchy"
   resultId="methodMatches" />
<assign id="aggregate" valueId="methodMatches"/>
<assert name="AggregateIsEmpty"/>

<!-- Commit (7) -->
<refactor name="CreateMethod"/>

<!-- Create the invocation of the new method (8a) -->
<body
   id="invocationBody"
   parentId="parent"/>

<!-- Marshall the invocation's arguments (8a-i) -->
<body.aggregate id="arguments"/>

<!-- Iterate through previously computed new method parameters and add the -->
<!-- name of each to the invocation as an argument -->
<iterate aggregateId="parameters">
   <assign id="parameter" valueId="parameters"/>
   <compute name="GetParameterName" resultId="argumentName"/>
   <body
      id="argument"
      contentsId="argumentName"
      parentId="parent"/>
   <assign id="actor" valueId="argument"/>
   <assign id="aggregate" valueId="arguments"/>
   <compute name="AddActorToAggregate" resultId="arguments"/>
</iterate>

<!-- Add invocation (method name + arguments) to the invocationBody (8b) -->
<assign id="name" valueId="newMethodName"/>
<assign id="body" valueId="invocationBody"/>
<compute name="AddInvocationTextToBody" resultId="invocationBody"/>

<!-- If extractionBody returns modified variable value then add an assignment -->
<!-- of the invocation to a local variable (9) -->
<assign id="actor" valueId="modifiedLocal"/>
<cond name="ActorIsNull" modifier="not">
  <assign id="variable" valueId="modifiedLocal"/>
  <assign id="body" valueId="invocationBody"/>
  <compute name="AddAssignmentToBody" resultId="invocationBody"/>
  <assign id="body" valueId="upperBody"/>
  <!-- If variable is declared in extractionBody then add a local -->
  <!-- declaration (10) -->
  <cond name="BodyDefinesLocalVariable" modifier="not">
    <assign id="body" valueId="extractionBody"/>
    <compute name="DetermineVariableType" resultId="type"/>
    <assign id="body" valueId="invocationBody"/>
    <compute name="AddLocalVariableDeclaration" resultId="invocationBody"/>
  </cond>
</cond>
</cond>

<!-- Set the body of the original method to the concatenation of upper body, -->
<!-- invocation body and lower body (11a) -->
<assign id="body" valueId="upperBody"/>
<compute name="GetBodyContent" resultId="string1"/>
<assign id="body" valueId="invocationBody"/>
<compute name="GetBodyContent" resultId="string2"/>
<compute name="AddStrings" resultId="string1"/>
<assign id="body" valueId="lowerBody"/>
<compute name="GetBodyContent" resultId="string2"/>
<compute name="AddStrings" resultId="newMethodBody"/>

<!-- Commit (11b) -->
<assign id="method" valueId="parent"/>
<body
   id="body"
   parentNode="parent"
   contentId="newMethodBody">
<refactor name="SetMethodBody"/>
</refactoring>

Figure 14. Expression of the ExtractBody refactoring
<xml version="1.0" encoding="UTF-8"/>
<!DOCTYPE assertion SYSTEM
 "http://www.scs.carleton.ca/~ggayed/refactoring.dtd">

<assertion name="CanExtractBlock">

<!-- List the actors that must be supplied -->
<requires id="fullBody" entity="body"/>
<requires id="extractionBody" entity="body"/>

<!-- Perform the assertion's steps -->

<!-- Split original body into upperBody, extractionBody and lowerBody (1) -->
<number id="start" value="0"/>
<-- upperBody (1b) -->
<assign id="body" valueId="extractionBody"/>
<compute name="GetBodyPosition" resultId="end"/>
<assign id="body" valueId="fullBody"/>
<compute name="ExtractBodyRange" resultId="upperBody"/>
<-- lowerBody (1c) -->
<assign id="number1" valueId="end"/>
<assign id="body" valueId="extractionBody"/>
<compute name="GetBodyContent" resultId="string"/>
<compute name="DetermineStringLength" resultId="bodyLength"/>
<assign id="number2" valueId="bodyLength"/>
<compute name="AddNumbers" resultId="lowerBodyStart"/>
<assign id="start" valueId="lowerBodyStart"/>
<assign id="body" valueId="fullBody"/>
<compute name="GetBodyContent" resultId="string"/>
<compute name="DetermineStringLength" resultId="end"/>
<compute name="ExtractBodyRange" resultId="lowerBody"/>

<!-- Ensure that at most one local variable is modified in -->
<!-- extractionBody and accessed in lowerBody (2) -->

<assign id="body" valueId="extractionBody"/>
<compute
    name="DetermineAccessedLocalVariables"
    resultId="localVariables"/>

<!-- Iterate through extractionBody's modified local variables (2b) -->
<!-- Increment counter for each that is referenced in lowerBody -->
<actor id="modifiedLocal"/>
<number id="counter" value="0"/>
<iterate aggregateId="localVariables">
  <assign id="body" valueId="extractionBody"/>
  <assign id="variable" valueId="localVariables"/>
  <cond name="BodyAssignsLocalVariable">
    <assign id="body" valueId="lowerBody"/>
    <cond name="BodyAccessesLocalVariable">
      <!-- Such a variable was found -->
      <assign id="number" valueId="counter"/>
      <compute name="IncrementNumber" resultId="counter"/>
      <assign id="modifiedLocal" valueId="variable"/>
    </cond>
  </cond>
</iterate>

<!-- Assert that no more than one such local variable was detected (2c) -->
<number id="threshold" value="2"/>
<assign id="number" valueId="counter"/>
<assert name="NumberLessThan"/>

<!-- Ensure that extractionBody will compile in isolation (3) -->
<assign id="body" valueId="extractionBody"/>
<compute
  name="DetermineAccessedLocalVariables"
  resultId="localVariables"/>

<!-- Add temporary variable declarations to extractionBody for local variables -->
<!-- that are accessed in extractionBody but declared in upperBody (3b) -->
<iterate aggregateId="localVariables">
  <assign id="body" valueId="extractionBody"/>
  <assign id="variable" valueId="localVariables"/>
  <cond name="BodyDefinesLocalVariable" modifier="not">
    <!-- Add a local variable declaration temporarily -->
    <assign id="variable" valueId="localVariables"/>
    <assign id="body" valueId="fullBody"/>
    <compute name="DetermineVariableType" resultId="type"/>
    <assign id="body" valueId="extractionBody"/>
    <number id="position" value="0"/>
    <compute
      name="AddLocalVariableDeclaration"
      resultId="extractionBody"/>
  </cond>
</iterate>
</iterate>

<!-- Compile (3c) -->
<assign id="body" valueId="extractionBody"/>
<assert name="BodyCompiles"/>

<!-- Ensure that original body will compile without extractionBody (4) -->

<!-- Check for variable declared in extractionBody and accessed in -->
<!-- lowerBody (4a) -->
<assign id="actor" valueId="modifiedLocal"/>
<cond name="ActorIsNull" modifier="not">
  <!-- Determine if variable is declared in upperBody (4b) -->
  <assign id="body" valueId="upperBody"/>
  <assign id="variable" valueId="modifiedLocal"/>
  <cond name="BodyDefinesLocalVariable" modifier="not">
    <!-- Add a local variable declaration temporarily (4c) -->
    <assign id="body" valueId="extractionBody"/>
    <assign id="variable" valueId="modifiedLocal"/>
    <compute name="DetermineVariableType" resultId="type"/>
    <number id="position" value="0"/>
    <assign id="body" valueId="lowerBody"/>
    <compute
      name="AddLocalVariableDeclaration"
      resultId="lowerBody"/>
  </cond>
</cond>
</cond>

<!-- Combine upperBody and lowerBody (4d) -->
<assign id="body" valueId="upperBody"/>
<compute name="GetBodyContent" resultId="upperBodyContents"/>
<assign id="body" valueId="lowerBody"/>
<compute name="GetBodyContent" resultId="lowerBodyContents"/>
<assign id="string1" valueId="upperBodyContents"/>
<assign id="string2" valueId="lowerBodyContents"/>
<compute name="AddStrings" resultId="fullContentsWithoutBlock"/>

<!-- Compile (4e) -->
<assign id="actor" valueId="fullBody"/>
<compute name="GetParent" resultId="parent"/>
<body
  id="body"
  contentsId="fullContentsWithoutBlock"
  parentId="parent"/>
<assert name="BodyCompiles"/>
</assertion>

Figure 15. Expression of the CanExtractBlock assertion
The tool defines an Eclipse extension point through which all XML-based operations are registered; this allows other plugins to register their own desired operations by defining extensions to this point. Eclipse's contained XML SAX parser performs syntactic validation of the registered operation expressions and parses the operations into elements that are then used by the tool to build internal representations of the operations. All operations are stored in a table for future reference.

4.2.1 Tool Implementation Details

The two primary functions performed by the tool are parsing of operations expressions into an internal representation and application of these operations as specified by the refactoring model. The tool's internal representation of operations is discussed first since the parsing and application components are dependent upon it. Following this, the object interactions that occur during the parsing and application of operations are described and shown.

The tool represents each operation internally as an instance of class Operation. Among other things, these instances act as containers for their respective steps. Every operation step is represented as a distinct object, as is prescribed by the Command pattern [GHJV95]. While each step's class is dependent upon its step type (e.g.-ActorDeclaration, Assignment, etc.), all steps implement common interface IStep, thereby allowing clients to treat them homogeneously. This approach greatly simplifies the parsing and application of refactorings.

Eclipse uses a SAX XML parser instance for parsing plugin point extensions such as operation expressions. The refactoring tool retrieves this parsed markup from the
Eclipse extension registry and processes it linearly. For each element that is encountered the tool creates a factory [GHJ95] whose purpose is to eventually create the Operation or IStep corresponding to this element once all sub-element processing has been completed. While the specific type of factory created is dependent upon the element encountered, all factories implement common interface IFactory, thus allowing them to be treated homogeneously. The factory is pushed onto a stack while the processing of sub-elements continues, which is an appropriate storage structure to use given the well-formed nature of valid XML expressions. The described factory creation steps are shown in Figure 16. When all sub-element processing is complete the factory is popped from the stack and queried for its completed Operation or IStep instance, which is then passed to the next factory on the stack. This mechanism ensures that created Operations and ISteps are aware of all contained sub steps. The described object creation steps are shown in Figure 17. The homogeneous nature of operation steps makes the application of non-primitive operations very straightforward. Application of a given operation is performed by the ExecutionFrame, which simply executes each of the operation's steps in sequence. Each step then applies itself appropriately according to the type of step that it represents. These operation application steps are shown in Figure 18. Figure 19 shows an object model diagram with the full set of types, and their relevant interfaces, which participate in the application of operations.
Figure 16. Operation expression start element interactions
Figure 17. Operation expression end element interactions
Figure 18. Operation application interactions
Figure 19. Operation application object model diagram
4.2.2 Tool Usage

An Eclipse user that wishes to use the tool to apply a defined refactoring starts the process by selecting the program element(s) in the Eclipse environment that are to be acted upon. For example, a user that wants to extract part of a method body into a new method would open the source method for editing and select the block of code to be extracted in the method editor. The user can then invoke the refactoring tool through the "Thesis" menu item that is added to Eclipse’s Refactoring Java perspective menu. A screen snapshot of the subsequent refactoring selection dialog is shown in Figure 20, which lists all registered refactorings. This dialog lets the user select the refactoring that is to be applied to the current program element selection. The selected refactoring’s stated actor requirements are used to bind the environment’s currently selected program elements to its required actor bindings. While this mechanism can be error-prone in the general usage case, it is adequate given the limited scope of the current prototype; this shortcoming is addressed in the final chapter. If appropriate actor bindings cannot be determined then the refactoring is not invoked.

Assuming that appropriate actor bindings are determined, the selected refactoring’s steps are then applied to the selected program element(s) as prescribed by the refactoring model. All output indicating progress of the application of the refactoring is written to Eclipse’s Console view. If a problem occurs during the application process, such as an assertion failure, then a dialog is shown to inform the user of the problem and no transformations are applied to the program elements in the environment.
Figure 20. Screen snapshot of the developed refactoring tool
4.2.3 Applicability to Other Languages

As a result of fulfilling Abelson and Sussman's criteria for expressive programming languages, the refactoring model can be used to express refactorings for any target language for which reasonable language entity to actor model mappings can be established. Java is presented here as a sample target language since the developed tool was written to act upon Java programs. However, the only change required for making the tool act upon programs written in another language, such as C++, is the substitution of corresponding C++ primitive operations in place of the existing Java primitives.

Figure 21 shows an example of primitive refactoring CreateClass, which is written to target the Java language. The interesting portion of this implementation is the bottom half of the code that follows the "(Java-specific portion follows)" comment, since this is where Eclipse's Java language bindings are utilised to perform work. A developer wishing to implement an equivalent primitive operation for acting upon C++ programs in Eclipse's CDT plugin could do so by copying the top portion of this primitive and reimplementing the bottom language-specific portion to connect with and act upon the available CDT model.

4.3 Summary

This chapter has presented two example refactorings that utilize the presented refactoring model and expression syntax. It has also described a developed tool that is able to automatically apply these refactorings in the Eclipse IDE. While the example refactorings do not serve as a formal proof of the completeness of the refactoring model,
package thesis.prim.refactorings;

import org.eclipse.core.resources.IWorkspaceRoot;
import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.jdt.core.***;
import org.eclipse.jdt.internal.ui.JavaPlugin;
import thesis.actors.***;
import thesis.executionunit.***;
import thesis.core.***;

public class CreateClass extends PrimitiveRefactoring {
    public CreateClass() {
        super("prim-createClass");
    }

    public boolean execute(ExecutionFrame frame) {
        // shared resources
        IProgressMonitor monitor = getProgressMonitor();
        String targetProjectName = getTargetProject();

        // separate class name into package and class components
        Type classToCreate = (Type)frame.getActorBinding("class");
        String fullClassName = classToCreate.getName().toString();
        int lastSeparatorIndex = fullClassName.lastIndexOf("");
        String packageName = "";
        if (lastSeparatorIndex != -1) {
            packageName = fullClassName.substring(0, lastSeparatorIndex);
        }
        String className = fullClassName.substring(lastSeparatorIndex + 1);

        /*
         * (Java-specific portion follows)
         * connect to Java model and attempt to create new class
         */
        try {
            IWorkspaceRoot workspace =
                JavaPlugin.getWorkspace().getRoot();
            IJavaModel model = JavaCore.getActiveModel().create(workspace);
            IJavaProject project = model.getJavaProject(targetProjectName);
            IPackageFragmentRoot root =
                project.getPackageFragmentRoots()[0];
            IPackageFragment pack =
                root.getPackageFragment(packageName);
        } catch (IOException e) { /* handle error */ }
    }
}
ICompilationUnit cu = pack.createCompilationUnit(
    className + ".java",
    packageName.equals("") ? "":
    "package " + packageName + "\n\n",
    false,
    monitor);
cu.createType(
    "public class " + className + " {\n}\n",
    null,
    false,
    monitor);
} catch (JavaModelException e) {
    e.printStackTrace();
    return false; // return failure indicator
}
return true; // return success indicator

Figure 21. Implementation of the CreateClass primitive refactoring for Java
their breadth of use of the model does provide validation with respect to its abilities and breadth. The developed tool demonstrates that the automated application of arbitrary refactorings can be performed given an appropriate refactoring model and means of expression. These results provide confidence in the work that was presented in the previous chapter.
Chapter 5

Conclusion

As was emphasized in the introduction, the cost savings that can be achieved by applying refactorings in an automated manner to software under development are enormous. Many refactoring studies and tool development efforts have been performed in an effort to make these benefits more easily realizable by developers. While these works have successfully encouraged the adoption of refactoring practices by many developers, their concrete uses have been limited to the automated application of pre-defined refactorings that are relatively low-level in nature. This thesis addressed the current inability of developers to apply arbitrary object-oriented refactorings to software in an automated manner. This situation leaves a developer that wishes to apply a refactoring that is not supported by an existing refactoring tool with only two options. These options are to either apply the desired refactoring to the software manually by hand and risk introducing new problems in the process, or not apply the refactoring at all.

The goal of thesis was to specify a means by which a developer could define and repeatedly apply any valid refactoring to software in an automated manner. This goal was achieved by specifying both a refactoring model that could represent the application of any valid refactoring, and a means by which such refactorings could be concretely expressed. These facilities are adequate for achieving the stated goal since they specify a set of refactoring concepts and rules that can be shared between a refactoring definer and
an automated refactoring application tool, and provide a means for communicating refactoring definitions in terms of this model from a refactoring definer to such a tool.

The contributions of this thesis are the concrete facilities that were developed in order to fulfill these goals. The first contribution was creation of the formal refactoring model. This model's refactoring definition concepts essentially boiled down to an actor model that was used to represent refactoring participant entities, an operation model that was used to describe the refactoring steps to be applied to these actors, and an execution model that dictated rules for the application of refactoring instances. The second contribution of this thesis was the specification of a formal XML-based syntax for expressing instances of this refactoring model. The specified syntactic rules are shown in Appendix A. The third contribution was the implementation of a tool that utilized these other contributions in order to facilitate the automatic application of refactoring model instances to software, within certain constraints. The tool was implemented such that it acted upon Java resources in the Eclipse IDE, but could easily be extended to work with resources from other languages in other environments.

5.1 Future Model Work

In assessing the use cases for this research, one refactoring definition aspect that fell outside of the scope of this thesis was the specification of standard sets of primitive refactorings, assertions and computations. The inclusion of standard primitives in the refactoring model would serve to extend its scope from the representation of refactoring logic to also prescribing a sufficient set of operations with which all refactorings can be
composed. This extension of the model is a requirement for ensuring that defined refactorings remain compatible amongst various refactoring application tools targeting various object-oriented languages. While the refactoring model has been specified such that developers can add primitive operations as needed, a refactoring defined solely in terms of these standard primitive operations and their compositions would consequently be universally applicable without change. The challenge is to determine this set of primitive operations appropriately such that they can be used to compose any refactoring that is applicable across all object-oriented languages. Determining these operation sets for manipulating higher-level actors such as Types and Methods is not difficult since the range of potential actions applicable to these types of actors is limited. Similarly, specifying an adequate set of aggregation manipulation primitive operations is also straightforward. However, determining the appropriate set of primitive operations for manipulating Body actors while utilizing solely language-independent concepts is more challenging. Once these primitive operations have been identified and formally specified then there will no longer be a need for developers to define additional non-standard primitive operations.

In addition to universally applicable sets of primitive operations, sets of standard language-specific primitive operations should also be specified in order to facilitate the definition of refactorings that utilize language-specific features. While refactorings composed with any of these primitive operations would not be usable by refactoring tools targeting other languages, they would at least be guaranteed to be usable by tools
targeting the specific language. Such a scenario is desirable for refactorings that are inherently only applicable across a subset of object-oriented languages.

Extending these notions, the specification of universal sets of composite operations would provide higher-level operations for developers to define their refactorings in terms of. While these composite operations would not enable the definition of any new refactorings beyond those that would already be definable using the universal primitive operations, they would greatly reduce the amount of work required to define new refactorings. This would especially be the case if the composites were carefully chosen so as to have wide applicability. One example of such a composite operation that could be useful in various contexts is the defined \texttt{MatchMethodSignatureInHierarchy} computation, which computes all methods in a type hierarchy that share a given signature. This computation was used in both the \texttt{AddParameter} and \texttt{ExtractMethod} refactorings shown in Chapter 4. Many refactoring definers could also benefit from the specification of standard language-specific composite operations.

\section{5.2 Future Model Expression Work}

While the presented expression syntax is adequate for expressing instances of the refactoring model, alternative formats may also be defined for accomplishing this task. For example, XSTL [Tid01] is a document transformation language that could be used in this context. XSLT style sheets are expressed as XML documents that must adhere to the formal XSLT specification. The specification includes a full set of tags for the
expression of generic processing steps, and therefore meets the requirements for expressing instances of the presented refactoring model. As such, it would be beneficial to explore implementation of refactoring model expressions as XSLT documents since this would allow refactoring definers to leverage an existing XML expression standard instead of having to use the new format.

5.3 Future Development Work

While the refactoring model dictates how refactoring instances are to be applied to existing software, it does not consider details of how a refactoring application tool should be implemented or used. This is purposely omitted because tool details fall outside of the scope of the refactoring model. Compliant tool implementations are free to provide any level of desired functionality to the user, provided that it meets the refactoring model standards at a minimum. The prototype tool that was developed as part of this research does not provide additional user facilities beyond those required for applying refactoring model instances, due to implementation time constraints. However there are many facilities that could be added to it that would increase its power and ease of use. Some of these facilities are discussed next, in order to both illustrate envisioned ultimate use for this research and in some cases to prescribe facilities that refactoring tools should provide in general.

One problem with the current prototype, as was mentioned in Chapter 4, is its ad hoc mechanism for mapping Eclipse IDE elements to tool actors. While this approach is often adequate for performing these mappings, it clearly cannot handle all use cases
properly, particularly for refactorings that have multiple required actors that are of the same entity type. The prototype should be extended so as to allow these mappings to be explicitly provided by the user whenever an ambiguous mapping situation occurs. This would ensure that selected refactorings are applied to the user's software as they intend. Furthermore, the tool could reduce the likelihood of encountering ambiguous mapping situations by only making refactorings available for selection whose actor requirements can be satisfied by the Eclipse elements selected at the time.

An important facility for encouraging users to apply refactorings in a more aggressive manner is the provision of Undo and Redo mechanisms. It is common for a user to have multiple refactorings available to choose from that could potentially improve their program's design, and selecting the refactoring that provides the maximum benefit is not always an obvious task. Allowing users to back out of refactorings that have been applied enables them to try various refactorings on a given program component in order to determine the one that best meets their needs. A complimentary facility that also enhances user confidence is a preview mechanism that shows the refactoring changes that are to be applied to a program before they are committed. This preview step gives the user the opportunity to change their mind about applying the refactoring before the actual changes have been performed. The current refactoring facilities that exist in the Eclipse environment provide a good example of how this should work.

An aspect of the presented refactoring model that can be problematic is the emphasis that it places on a refactoring definer for ensuring its behaviour preservation property. This approach was adopted because proving behaviour preservation for a given
transformation can be extremely difficult for a tool to do well, and would likely require
the imposition of some inappropriate constraints on refactorings. However, placing this
responsibility on the developer introduces the risk of allowing non-behaviour preserving
transformations to be considered as valid if the developer does not perform a proper
upfront analysis. A refactoring application tool could be instrumented to detect at least
some situations where behaviour is obviously not being preserved by a utilized
refactoring and alert the tool user of such cases. This would provide an additional degree
of confidence to developers using defined refactorings, and in particular those
refactorings that have been defined by others.

Lastly, the composing of refactoring expressions by hand can be a long and
tedious process. A tool that enabled a developer to define a refactoring by some means
other than writing XML could make the definition of refactorings much easier. One
example of such a facility would be a graphical tool that allowed developers to
communicate required transformations by rearranging entities in UML diagrams as
appropriate. The tool would then be responsible for observing these steps and generating
a corresponding refactoring model expression. While such a facility is not something that
all refactoring tools should necessarily offer, the existence of such a tool would greatly
encourage developers to define many more custom refactorings as a result of the reduced
effort that would be required. In general, any step that can be taken to encourage
developers to define useful new refactorings should be given serious consideration since
the benefits that can be realized from this research increase according to the pool of
publicly available defined refactorings.
5.4 Summary

The ability to define complex custom refactorings for automated application can make the use of refactorings even more straightforward and powerful. It should be noted that the ability to define a program refactoring in a standard way could be useful in contexts beyond those presented in this thesis. For example, tools performing either design analysis or adaptive programming could express refactoring suggestions in terms of the model syntax, for subsequent application by a tool. This means of facilitating tool interoperability makes the prospective usage contexts for refactoring application tools quite vast. The potential uses of the research in this thesis are limited only by the variety of useful tools that need a way to perform automated software refactorings.
Bibliography


[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.


<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
</table>
Appendix A

Refactoring Model Data Type Definition

<?xml version="1.0" encoding="UTF-8"?>

<!ENTITY % allActorTypes "access | access.aggregate | attribute | attribute.aggregate | body | body.aggregate | invocation | invocation.aggregate | method | method.aggregate | parameter | parameter.aggregate | type | type.aggregate | actor">

<!ENTITY % allTypes "%allActorTypes; | number | number.aggregate | string | string.aggregate | actor.aggregate">

<!ELEMENT access EMPTY>
<!ATTLIST access
  id CDATA #REQUIRED
  parentId CDATA #REQUIRED
  positionId CDATA #IMPLIED
  targetAttributeId CDATA #REQUIRED>
<!ELEMENT access.aggregate EMPTY>
<!ATTLIST access.aggregate id CDATA #REQUIRED>

<!ELEMENT attribute EMPTY>
<!ATTLIST attribute
  id CDATA #REQUIRED
  modifiersId CDATA #IMPLIED
  nameId CDATA #REQUIRED
  parentId CDATA #REQUIRED
  typeId CDATA #REQUIRED>
<!ELEMENT attribute.aggregate EMPTY>
<!ATTLIST attribute.aggregate id CDATA #REQUIRED>

<!ELEMENT body EMPTY>
<!ATTLIST body
  id CDATA #REQUIRED
  accessesId CDATA #IMPLIED
  contentsId CDATA #IMPLIED
  invocationsId CDATA #IMPLIED
  modifiersId CDATA #IMPLIED
  parentId CDATA #REQUIRED
positionId CDATA #IMPLIED>
<!ELEMENT body.aggregate EMPTY>
<!ATTLIST body.aggregate id CDATA #REQUIRED>

<!ELEMENT invocation EMPTY>
<!ATTLIST invocation
    id CDATA #REQUIRED
    parentId CDATA #REQUIRED
    positionId CDATA #IMPLIED
    targetMethodId CDATA #REQUIRED>
<!ELEMENT invocation.aggregate EMPTY>
<!ATTLIST invocation.aggregate id CDATA #REQUIRED>

<!ELEMENT method EMPTY>
<!ATTLIST method
    id CDATA #REQUIRED
    bodyId CDATA #IMPLIED
    modifiersId CDATA #IMPLIED
    nameId CDATA #REQUIRED
    parametersId CDATA #IMPLIED
    parentId CDATA #REQUIRED
    returnTypeId CDATA #IMPLIED>
<!ELEMENT method.aggregate EMPTY>
<!ATTLIST method.aggregate id CDATA #REQUIRED>

<!ELEMENT parameter EMPTY>
<!ATTLIST parameter
    id CDATA #REQUIRED
    modifiersId CDATA #IMPLIED
    nameId CDATA #REQUIRED
    parentId CDATA #REQUIRED
    typeId CDATA #REQUIRED>
<!ELEMENT parameter.aggregate EMPTY>
<!ATTLIST parameter.aggregate id CDATA #REQUIRED>

<!ELEMENT type EMPTY>
<!ATTLIST type
    id CDATA #REQUIRED
    nameId CDATA #REQUIRED
    supertypesId CDATA #IMPLIED
    methodsId CDATA #IMPLIED
    attributesId CDATA #IMPLIED
    modifiersId CDATA #IMPLIED>
<!ELEMENT type.aggregate EMPTY>
<!ATTLIST type.aggregate id CDATA #REQUIRED>

<!ELEMENT actor EMPTY>
<!ATTLIST actor id CDATA #REQUIRED>

<!ELEMENT number EMPTY>
<!ATTLIST number
    id CDATA #REQUIRED
    value CDATA #REQUIRED>
<!ELEMENT number.aggregate EMPTY>
<!ATTLIST number.aggregate id CDATA #REQUIRED>

<!ELEMENT string EMPTY>
<!ATTLIST string
    id CDATA #REQUIRED
    content CDATA #REQUIRED>
<!ELEMENT string.aggregate EMPTY>
<!ATTLIST string.aggregate id CDATA #REQUIRED>

<!ELEMENT requires EMPTY>
<!ATTLIST requires
    id CDATA #REQUIRED
    entity (%allTypes;) #REQUIRED>

<!ELEMENT assert EMPTY>
<!ATTLIST assert
    name CDATA #REQUIRED
    modifier (not) #IMPLIED>

<!ELEMENT compute EMPTY>
<!ATTLIST compute
    name CDATA #REQUIRED
    resultId CDATA #REQUIRED>

<!ELEMENT refactor EMPTY>
<!ATTLIST refactor
    name CDATA #REQUIRED>

<!ELEMENT assign EMPTY>
<!ATTLIST assign
    id CDATA #REQUIRED
    valueId CDATA #REQUIRED>
<!ELEMENT iterate (refactor | assert | compute | iterate | assign | cond | %allTypes;)*>
<!ATTLIST iterate aggregateId CDATA #REQUIRED>

<!ELEMENT cond (refactor | assert | compute | iterate | assign | cond | %allTypes;)*>
<!ATTLIST cond
  name CDATA #REQUIRED
  modifier (not) #IMPLIED>

<!ELEMENT refactoring (requires | refactor | assert | compute | iterate | assign | cond | %allTypes;)*>
<!ATTLIST refactoring
  name CDATA #REQUIRED>

<!ELEMENT assertion (requires | assert | compute | iterate | assign | cond | %allTypes;)*>
<!ATTLIST assertion
  name CDATA #REQUIRED>

<!ELEMENT computation (requires | assert | compute | iterate | assign | cond | %allTypes;)*>
<!ATTLIST computation
  name CDATA #REQUIRED>