© Copyright by Donald Bradley Roberts, 1999

PRACTICAL ANALYSIS FOR REFACTORING

BY

DONALD BRADLEY ROBERTS

B.S., University of Illinois, 1990
M.S., University of Illinois, 1992

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1999

Urbana, Illinois

PRACTICAL ANALYSIS FOR REFACTORING

Donald Bradley Roberts, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1999
Ralph Johnson, Advisor

One of the important ways to make software soft, (i.e., easy to change, reuse, and develop), is to automate the various program transformations that occur as software evolves. Automating refactorings is hard because a refactoring tool must be both fast and reliable, but program analysis is often undecidable. This dissertation describes several ways to make a refactoring tool that is both fast enough and reliable enough to be useful.

First, it gives a new definition of refactoring that focuses on preconditions and postconditions of the refactorings, rather than on the program transformation itself. Preconditions are assertions that a program must satisfy for the refactoring to be applied, and postconditions specify how the assertions are transformed by the refactoring. The postconditions can be used for several purposes: to reduce the amount of analysis that later refactorings must perform, to derive preconditions of composite refactorings, and to calculate dependencies between refactorings. These techniques can be used in a refactoring tool to support undo, user-defined composite refactorings, and multi-user refactoring.

This dissertation also examines techniques for using runtime analysis to assist refactoring, presents the design of the Refactoring Browser, a refactoring tool for Smalltalk that is used by commercial software developers, and identifies the criteria that are necessary for any refactoring tool to succeed.

To my wife, Denise and our child.

Acknowledgments

I am indebted to John Brant, the greatest Smalltalk hacker in the world, for all his contributions to this project and for keeping me honest. Without his insight and skill, the Refactoring Browser would never be solid enough for commercial use.

I would like to thank my advisor, Ralph Johnson, for many years of guidance and inspiration. He has constantly taken care of his grad students by pointing out relevant research, generating ideas, and finding work for them.

Many thanks to Brian Foote for his views on software evolution and for ever being the unsung champion of some of these ideas. Thanks to William Opdyke for pioneering the concept of formal refactoring and for sharing his experiences with the PhD process with me. I want to thank our research group for their feedback on my ideas, this thesis, and many laughs at the expense of methodologists.

I wish to thank the Fannie and John Hertz Foundation for providing the fellowship that funded a majority of this research.

I wish to especially thank my wife, Denise, for her love and support throughout this entire process and providing the much needed motivation by threatening to become a doctor before me.

Most importantly, I thank God for granting me the talents to achieve this accomplishment and for my salvation. Fundamentally, nothing else really matters.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Programs change. Usually, they change because of changing requirements or because the context in which they exist changes. But sometimes the changes to a program do not affect its behavior, only its internal structure. For example, a programmer might change the name of functions, or replace duplicate code with calls to a function, or change the interface of a function to make it more reusable. Sometimes programmers rewrite part of a program so they can understand it or even to avoid injuring their sense of aesthetics. In object-oriented programming, behavior-preserving source-to-source transformations such as these are known as **refactorings**. This dissertation is about how to automate refactorings.

To demonstrate the power of refactorings, we present an example that was taken from a real case. Smallwalker is a web browser that is implemented entirely in Smalltalk. Within this software package are several useful components such as classes that represent URLs, HTML entities, and several types of images. However, these components were not designed for reuse outside of the web browser. We wish to refactor the design of Smallwalker to make these components more separable.

Figure 1.1 shows a diagram of the classes that are used to represent a URL and fetch its contents from a server. There are a couple of serious problems with the design as it stands. First, it does not map to the model that we have in our heads as to what a URL is. Conceptually, a URL is a handle to a Stream. A client should be able to send the readStream message to a URL and have a simple stream returned. Second, every time we add a new type

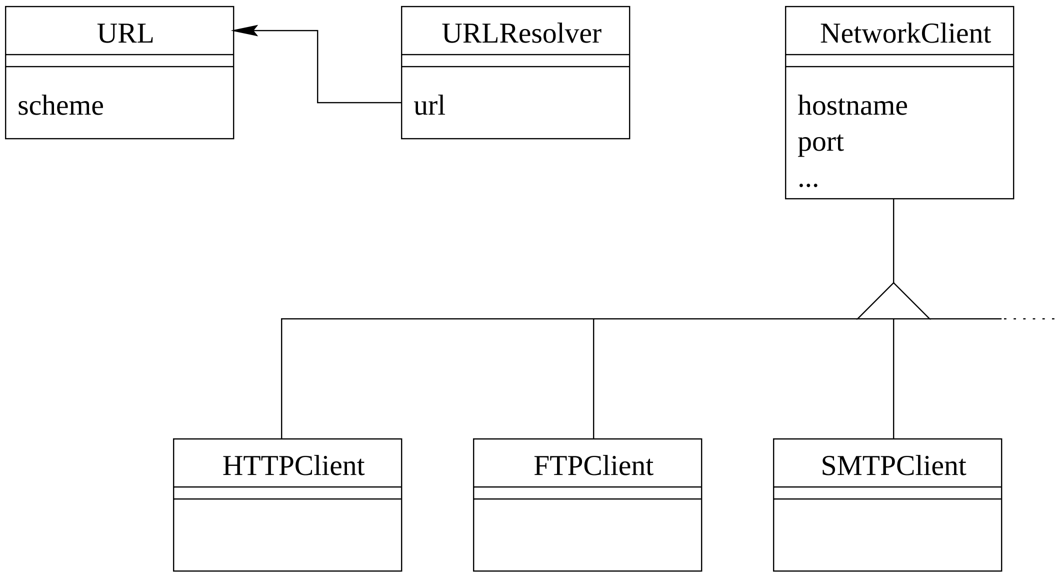


Figure 1.1: Original Design of the URL Classes

of URL to the system, we have to update the code in several places. This is most obvious in the code shown in Figure 1.2, which must be changed to test for the new type of URL.

Ideally, we would like to simply have a single hierarchy, rooted at URL with subclasses for each type. By sending the named: message to URL with an argument which is the string representing the URL itself, an appropriate subclass is instantiated by looking up the type of URL in a dictionary that maps it to the subclass that implements the correct protocol. This new URL object responds to the readStream message, which returns a Stream containing the contents of the location. Figure 1.3 shows this refactored hierarchy. If each subclass of URL registers itself when it is installed in the system, there is no need to change any code when a new type of URL is created.

The program should run the same as it did before we performed this transformation, so where's the value in it? One value is that we have a clean subsystem for representing URLs. This subsystem can be reused in any application that needs to access information on the World Wide Web. In fact, this URL hierarchy has a protocol that is similar to the Filename class, so URLs could possibly be used where only Filenames were allowed before. Another

```
URLResolver>>getContentsNoSignalReportTo: progressStatus noCache: noCache
    | scheme |

    scheme := url scheme.
    scheme = 'http' ifTrue: [↑self getHttpContentsReportTo: progressStatus noCache: noCache].
    scheme = 'ftp' ifTrue: [↑self getFtpContentsReportTo: progressStatus noCache: noCache].
    scheme = 'gopher' ifTrue: [↑self getGopherContentsReportTo: progressStatus noCache: noCache].
    scheme = 'wais' ifTrue: [↑self getWaisContentsReportTo: progressStatus noCache: noCache].
    scheme = 'mailto' ifTrue: [↑self getMailToContentsReportTo: progressStatus noCache: noCache].
    scheme = 'news' ifTrue: [↑self getNewsContentsReportTo: progressStatus noCache: noCache].
    scheme = 'nntp' ifTrue: [↑self getNntpContentsReportTo: progressStatus noCache: noCache].
    url isFileScheme ifTrue: [↑self getFileContentsReportTo: progressStatus noCache: noCache].
    ↑self errorContents: 'Unsupported protocol'
```

Figure 1.2: Code that is Difficult to Maintain

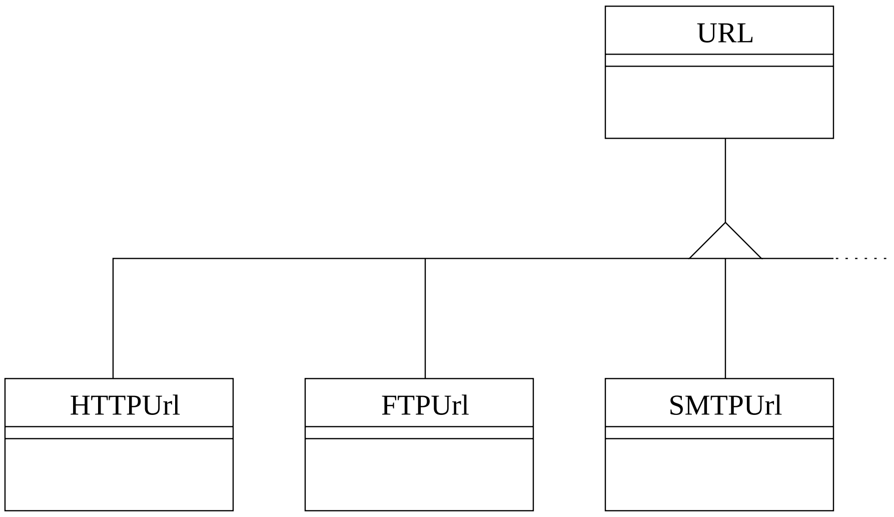


Figure 1.3: Refactored Design of the URL Classes

value is that the code is more maintainable. Adding new URLs no longer requires changing any existing code, reducing the chance of errors.

This is a fairly significant design change, but the code worked before, and should work after this refactoring. Even though this is a large transformation, it can be decomposed into a series of very small transformations. The goal of refactoring tool research is to provide support for these small refactorings. If these small refactorings can be shown to be correct, then large changes that are composed solely of small refactorings must be correct.

Refactoring is nothing new. Programmers have done it for years in many different languages, although it has been called by different names. It has typically been done manually, or with the help of primitive tools such as text editors with search and replace. Experienced programmers recognize that code should be "cleaned up" before submission to a system. This cleanup involves adding comments, renaming entities, and clarifying algorithms. However, manually performing these tasks is fairly time-consuming and programmers under time pressure will not do it. Similarly, experienced programmers are loathe to touch existing code that works for fear of introducing an error. As the southern aphorism goes, "if it ain't broke, don't fix it."

This situation leads to several problems in software: unclear code which leads to corrupted designs which leads to high maintenance costs. Unclear code arises because what the original developer considers "obvious" is often not obvious to the next programmer. As requirements change during the lifetime of the software, the design is not updated to encompass the new requirements since changing the design would involve refactoring large amounts of code. Therefore, changes are made in the most expedient means possible and the design of the system becomes increasingly corrupt and brittle.

These effects are due to the fact that manual refactoring is tedious and error-prone and therefore very costly. Like all tedious, error-prone, manual processes, it should be automated. It is obvious that tool support for refactoring will reduce maintenance cost by allowing the programmers to adjust the design to accommodate new features. But what is less obvious is its impact on the entire software development process.

Up-front software analysis and design methodologies assume that it is expensive to change the design once it has been realized in code. As rearranging the structure of code becomes cheaper, it becomes less costly to change the design of the software. As a result, it makes sense to spend less money on up-front design since shortcomings in the design can be corrected by refactoring the code later. This leads to a qualitatively different form of software development where the software is designed to support the immediate need and refactored later to add flexibility to support future requirements. One of red flags that signals an inadequate design is the complexity of the code necessary to realize it. If a particular design requires awkward coding or convoluted control flow to implement, its quality is suspect. However, in traditional software development methodologies, it is often too late to correct the design once a significant amount of code exists. The late-design approach creates designs that are no more or less complex than necessary to meet the current set of requirements. Since the designers do not need to attempt to predict the ways in which the system might evolve in the future, they create systems that are more maintainable and cheaper to build. One of the first proponents of this approach was Carl Hewitt who referred to this style of design as incremental perpetual development [Hew77]. Recently, this has been incorporated into a commercial development methodology known as Extreme Programming [GHH97][Bec99].

Refactoring occurs at all levels. There are high-level refactorings that are major design changes, and there are low-level refactorings such as renaming a variable. William Opdyke took the approach that high-level refactorings can be implemented in terms of several low-level refactorings [Opd92]. If we can implement the low-level refactorings correctly, then the high-level refactorings will be correct. This is the same view that this dissertation takes and much of our research is based directly on the groundwork he laid.

Our focus has been on developing tools to be used by programmers. Our approach is to create tools to assist a human in refactoring code rather than attempt to automatically detect better designs and refactor the code. We provide support for these activities in the tool, but always leave most of the control in the hands of the programmer.

1.1 Definition of Refactoring

One of the problems with automating traditional refactorings is that they must be behavior preserving. This property is difficult to prove and the analyses that must be performed to ensure this are often difficult to compute. In this dissertation, we present an alternative definition of refactoring that is easier to automate and reason about while still being useful to programmers. We accomplish this by moving the transformation out of a language that is difficult to reason about, (i.e., programs) into a more tractable language, (i.e., first order predicate calculus, in our case). This definition allows us to rapidly calculate interesting properties of sequences of refactorings.

As part of this research, we have built commercial-grade tools to support refactoring in the Smalltalk language. While there has been much research into source-to-source transformations, we know of no commercially successful tools that have been widely accepted by industry programmers. The Refactoring Browser currently has hundreds of users that use it on a daily basis to continually refactor their code. We feel that this is a huge leap forward for software engineering in that programmers trust the tool to regularly make global changes in their software. While making the tool successful we have identified several properties of tools that determine whether they will be used by programmers. This dissertation discusses the architecture of the tool along with criteria, both technical and practical, that are necessary for the success of a refactoring tool. Constructing this tool has presented several interesting problems that this dissertation will address. The problems this dissertation examines are: using postconditions to eliminate analysis, computing dependencies between refactorings, and using dynamically obtained properties of code to perform refactorings. We will briefly describe each of these problems.

1.2 Using Postconditions to Eliminate Analysis

In his dissertation, William Opdyke presented twenty-three primitive refactorings that he had identified by observing changes that object-oriented programmers made during the evolution of their code. He described each of the primitive refactorings and proved the preconditions that must be met to ensure that the transformation preserves program behavior. He then created three more complicated refactorings by composing the primitive refactorings. Since each primitive refactoring was behavior preserving, the composition was necessarily behavior preserving.

His primitive refactorings are rarely performed in isolation. They are too small. Refactorings are usually performed in a group. Whenever programmers restructure code, they start with a system and have some design goal in mind. For example, a typical design goal is “extract the common part of these two methods and move it up to the superclass.” They then perform several refactorings to produce a new system that achieves the design goal. The sequence of refactorings that solves this design goal can be thought of as a composite refactoring that is made up of a set of atomic refactorings.

The atomic steps of a composite refactoring are performed in strict sequence. Later refactorings often depend on earlier refactorings to put the system into a particular state. For example, consider refactoring an object that implements an algorithm internally to one that uses the Strategy pattern [GHJV95], which is the encapsulation of an algorithm within a single object. The programmer creates a new class to represent the algorithm, and moves the instance variables and methods that implement the algorithm within the original object into the new strategy object. Instance variables can only be moved into the new object if the new object and the original object exist in a one-to-one relationship everywhere in the program. This relationship between objects is extremely difficult to prove in the general case. However, since the programmer created the object and established the relationship with prior refactorings, we know that it will always exist in a one-to-one relationship.

This dissertation augments the definitions of the refactorings presented by Opdyke by adding postcondition assertions. We present several of his refactorings with these new definitions. Refactorings are typically applied in sequences specifically intended to set up preconditions for later refactorings. This observation along with our postconditions allows us to eliminate much of the more expensive analysis that is associated with ensuring that refactorings are legal.

1.3 Computing Dependencies Between Refactorings

As stated in the previous section, some refactorings in a sequence exist to satisfy the preconditions for later refactorings. Other times, the ordering within the sequence is arbitrary. An interesting and useful property to compute is whether one refactoring **depends** upon another. Informally, two refactorings do not depend upon one another if it doesn't matter what order they are applied to the program. Slightly more formally, a refactoring, R_2, depends upon another, R_1, if R_2 cannot be legally applied to a program unless R_1 has been applied first. Therefore, our definition of dependency will be based on whether two refactorings commute. If two refactorings can be commuted in a sequence, they do not depend on one another. We will present a formula to derive the conditions under which two refactorings commute. We will then give sample derivations of the dependency conditions between a couple of pairs of refactorings.

The dependency property has several uses in the development of refactoring tools. Some of the uses that we will describe in this dissertation are: undoing a refactoring within a sequence, detecting conflicts between two groups of refactorings, and parallelizing refactorings.

Our tool is an interactive tool and is often used in an exploratory manner. One of the key features of an exploratory tool is the ability to undo manipulations. This allows the programmer to refactor the code one way, see how it looks, and if it is not satisfactory, undo the refactoring. As noted before, sequences of refactorings are sometimes arbitrary. The tool can allow the programmer to select an arbitrary refactoring in the sequence and choose to undo it. By using dependency information, the tool can then undo the later refactorings in the sequence that depend on the refactoring the programmer wants to undo.

Programming projects are rarely accomplished by single programmers. When multiple programmers refactor the same program, there is a potential for conflicts. A refactoring tool can use dependency information to check two sequences of refactorings for conflicts. This can yield a finer-grain of conflict than standard source code control approaches.

As a system gets larger, refactoring takes longer. A large sequence of refactorings applied to a large system could be too slow to be practical. Dependency information can be used to determine which refactorings within the sequences *must* be performed sequentially, and which ones can be performed in parallel. By distributing the parallelizable refactorings to multiple processors, the sequence can be applied more quickly.

1.4 Using Dynamic Properties of Code

Automated refactorings have an obvious implementation. The designer decides which refactoring to perform, then the system statically analyzes the source code to determine if the refactoring's preconditions are satisfied. If the preconditions are not satisfied, the designer is notified and no action is taken. If the preconditions are satisfied, the system performs the refactoring.

This paradigm has both strengths and weaknesses. Since the analysis is static, it is necessarily a conservative approximation. Therefore, the refactoring is guaranteed to preserve the program's behavior. However, since the analyses are approximations, the conservative approach rules out legal refactorings. Some of these approximations are particularly poor, such as the analysis for object ownership presented by Opdyke which states that an object is **owned** by another only if it is created in the constructor of the owning object and is never referenced outside of the object [Opd92]. Analyses that are this restrictive disallow nearly all the applications of refactorings that depend on them. Even in the cases where useful static approximations can be computed, if these approximations take an unreasonable amount of time to compute, any system based on them will be unacceptably slow for an interactive tool.

As a solution to these weaknesses, I propose a radically new paradigm for refactoring. In this paradigm, the analyses for which it is difficult to compute reasonable, timely, static approximations are performed at runtime by observing the executing program. The results of these analyses are fed to refactorings that transform the running program or, if a particular refactoring is determined to be invalid, undoes itself and any refactorings that depend upon it.

This paradigm is not without its weaknesses. Since the refactorings are performed at runtime, the analysis is only as good as the test suite used to exercise the program. Paths that the test suite does not exercise will not get analyzed. This can possibly lead to errors that the user will uncover. Development of good test suites is known to be a difficult problem [Mar95], so we are not solving a hard problem, we are transforming it into a different hard problem. However, with measures such as coverage the test suite's quality, and therefore, the quality of the dynamic analysis, can be measured.

1.5 Contributions

The contributions of this research are:

1. A formal definition of refactoring that extends Opdyke's with postconditions.

2. Specification of several common refactorings with the new definition.

3. Design of a refactoring tool that reduces program analysis by using postcondition assertions.

4. A method of calculating the preconditions for composite refactorings.

5. Definition of dependency between refactorings based on commutativity.

6. A method for calculating the conditions under which any two refactorings may commute.

7. A set of applications that use the dependency information calculated from a chain of refactorings.

8. A scheme for using dynamically obtained information to perform refactorings.

The next chapter surveys the previous work in this and related areas as well as provides the context for the types of tools we want to build. Chapter 3 provides the formal definitions of the refactorings and analysis functions that we will be studying. Chapter 4 derives the conditions under which one refactoring is dependent upon another and gives some sample derivations of that property. Chapter 5 describes refactoring using runtime information. Chapter 6 discusses the requirements to make a refactoring tool succeed. Chapter 7 presents the architecture of the refactoring tool that we have built. Chapter 8 summarizes the contributions of this research and describes future work that this research points to.

Chapter 2

Related Work

Program transformation has been studied in several different contexts. It was originally studied as an approach to implementing interpreters, compilers, and optimizers with transformations such as loop unrolling and constant propagation [ASU88]. However, more recently it has been used to support programmers in their work.

2.1 Program Transformation

Research in source-to-source transformation is more rare and has generally been studied in the area of partial evaluation of programs [CD93] and in term rewriting systems. Term rewriting is an approach to program development where the program is specified as a set of recursive equations that are transformed into an efficient implementation of the program. Each transformation is shown to preserve the behavior of the equations, so the final program is equivalent to the original specification [BD77, KB70, DR93]. The main shortcoming to this technique is that it only applies to functional languages and therefore will require further processing to convert the program into a conventional programming language with imperative operations and destructive update [Fea82].

2.2 Program Maintenance

Much of the early work in program restructuring was inspired by the need to reduce the cost of maintaining programs. It has been shown that the principle cost of any software development is the maintenance after the software is "done." A study of one Air Force system revealed that it cost \$30 per line to develop and \$4,000 per line to maintain over its lifetime [Boe75]. By analyzing the IBM OS/360 project, Belady and Lehman determined that the cost of a change rose exponentially with respect to a system's age. They attributed this rising cost to the decay of the software's structure [BL71, BL76]. Gerald Weinberg maintains a private list of the world's most expensive program errors. The top three errors involved the change of exactly one line of code [Wei83]. His theory as to why these errors happened is that since the actual change was so small, the programmers did not take the time to fully test the code or consider the ramifications of the change.

This expenditure on maintenance has driven research into tools to support this process. One of the techniques that has been explored to reduce this effect is program slicing [Wei79, Wei84, Wei82]. A program slice, $S(v, n)$, on variable v at statement n yields the portions of the program which contributed to the value of v immediately before statement n is executed. This slice is itself and executable program. This slice of the original program can be compiled and executed in a debugging environment to allow the programmer to see *only* the code that can affect the variable in the line that he is interested in. By using an extension of this concept, Gallagher and Lyle create a *decomposition slice* that gives every statement in a program that affects the computation of a variable. By using this decomposition slice, maintainers can immediately see which statements of the program will be affected by a change [GL91]. Most of the applications of program slices have been to increase program comprehension for some task, be it debugging, maintenance, or re-engineering legacy systems.

William Griswold and his students have been studying tools to assist in the program restructuring process. He has tried to provide support not only for comprehending programs, but also manipulating them. His initial versions worked with the Scheme programming language [Gri91, GN93]. His tools used a control flow graph (CFG) and a program dependency graph (PDG) as the central representations for manipulating the code. His transformations correspond very closely with the method-level refactorings that we consider. To help visualize the program during maintenance, he used the star diagram[BG94, Bow95, BG98], a graphical view of the central data structures, that could be manipulated by the programmer to perform restructurings. Many of the transformations that Griswold and Bowdidge examined were below the function level, such as extracting portions of a function into a new function. Our research focuses on the higher-level entities of the systems such as classes and instance variables.

An area of research that is related to refactoring object-oriented systems, but which has surprisingly little overlap is the area of transforming and maintaining abstract data types [Sch81, Sch86, Sch94]. Scherlis' techniques allow programmers to alter the encapsulation boundaries of abstract data types while preserving the meaning of the program. His goal is to allow the programmer to adjust these boundaries to change the design of an existing system. He refers to this approach as *fluid architecture*. This concept is similar to the ideas of evolutionary software development that we will discuss in Section 2.4.

2.3 Refactoring

In object-oriented systems, behavior-preserving transformations are known as *refactorings*. This term probably originates from a quote by Peter Deutch that "interface design and functional factoring constitute the key intellectual content of software and are far more difficult to create or recreate than code [Deu89]." If separating function into objects is factoring, then changing where the function exists must be *re*factoring [OR95].

It has been recognized that reusable components are not designed correctly the first time, but come about by creating concrete applications and then abstracting the common parts

[JF88, RJ98]. The reason for this is that people think concretely much better than they think abstractly. When attempting to create abstractions from scratch, people will make some parts more general than necessary and fail to generalize other parts that should be. By examining several concrete applications, common parts become apparent much more quickly, even if the common parts are not identical. When they are not identical, the programmer can observe *exactly* how they vary and generalize to support that variance. Several researchers have developed program manipulation techniques to support this activity.

The Demeter system was a tool to facilitate the evolution of a class hierarchy [LHR88]. This system sought to enforce a notion of good object-oriented style which has come to be known as the Law of Demeter. The system provided an algorithm that would transform programs that did not conform to the law to ones that did. They go on to prove that any program that is written in a fashion that does not conform to the Law of Demeter can be rewritten to conform.

To support this algorithm, the Demeter system introduced a data structure known as the *class dictionary graph*. The vertices of the graph are classes and members. There are two types of edges that represent the relationships between two vertices, alternation edges, which represent inheritance relationships, and construction edges, which represent class/member relationships.

Bergstein used the class dictionary graphs to define a set of object-preserving class transformations [Ber91]. He defined a set of primitive, object-preserving transformations, and then derived many higher level transformations based upon them. His transformations were based mainly on inheritance.

Casais identified four categories of support for evolution within object-oriented systems:[Cas91]

Tailoring is the process of slightly adapting a class definition without subclassing. This category includes manipulations such as renaming, redefinition of attributes, or access privileges.

Surgery is the process of adding or removing entities from the program. Surgery includes adding or deleting classes, adding or deleting attributes.

Versioning is the process of tracking the changes in the individual entities within the system.

Reorganization is the process of making large changes to the class library. During this process, the programmers will try alternative designs.

Refactoring supports all of these behaviors with the possible exception of versioning. Casais developed an algorithm that would introduce new classes into a hierarchy to hold common attributes [Cas91]. A similar approach was taken by Ivan Moore, only at a finer granularity. Moore worked with the Self language[US87] and developed a tool called Guru that would extract common portions of methods into a new method which was then pulled up into a newly introduced class. His approach ignored the existing inheritance structure and then recreated an inheritance structure by finding common portions of methods, creating new methods that contained the common code, and then creating an inheritance hierarchy that allowed the common code to be inherited [Moo96].

The principle flaw with both of these approaches, which Moore acknowledges, is that they introduce new entities that do not necessarily correspond to any design element. Additionally, since the new entities are being created by the system, they must be assigned names in some automatic fashion. Figure 2.1 shows an example of the type of methods Guru creates. Since one of the principle reasons to refactor code is to increase its comprehensibility, we feel that this tool, while interesting theoretically, is not useful in a production code environment. Another drawback is that Moore's algorithm was too slow for an interactive tool, taking over 2 hours for a hierarchy of 17 classes with 366 members. By way of comparison, the entire VisualWorks image has over 1,000 classes and over 22,000 methods [Obj98]. Despite these problems, these algorithms could be useful in an interactive tool by restricting the scope of analysis and allowing the programmer to choose names for newly introduced entities.

```
newMethod651P: a P: res = (res sign: (sign * (a sign)))
newMethod627 = (-1 = sign)
newMethod659P: d = ((d size) - cByteSize)
mostSignificantDigit: d = (in d At: (newMethod659P: d))
newMethod636P: digit = ((digit asByte) - ('0' asByte))
newMethod661 = (size + 1)
newMethod695 = (size: (newMethod661))
```

Figure 2.1: Automatically Generated Method Names in Guru [Moo96]

Most of this research is based directly on William Opdyke's dissertation. In his dissertation, he identified the 23 primitive refactorings shown in Table 2.1 and gave examples of three composite refactorings [Opd92]. For each primitive refactoring, he defined a set of preconditions that would ensure that the refactoring would preserve behavior. He arrived at his collection of refactorings by observing several systems and recording the types of refactorings that OO programmers applied. His refactorings were defined in terms of C++, but many of them are applicable to other OO languages. The only refactorings that are not applicable to Smalltalk are the ones dealing with types and access privileges, since Smalltalk does not have these features.

This dissertation extends his ideas in several directions. First, correctness is a difficult property to prove. Opdyke identified seven program properties that his refactorings preserved:

1. *Unique Superclass*—Every class must have exactly one superclass. Even though his research focused on C++, he only considered single inheritance systems.

2. *Distinct Class Names*—Every class in the system must have a unique identifier. Even in the presence of nested scopes or namespaces, this property must be true.

3. *Distinct Member Names*—This property enforces distinct member names within a single class. The method can still be redefined in either superclasses or subclasses.

4. *Inherited Member Variables Not Redefined*—Classes cannot define variables that are

creating an empty class
creating a member variable
creating a member function
deleting an unreferenced class
deleting an unreferenced variable
deleting a set of member functions
changing a class name
changing a variable name
changing a member function name
changing the type of a set of variables and functions
changing access control mode
adding a function argument
deleting a function argument
reordering function arguments
adding a function body
deleting a function body
convert an instance variable to a variable that points to an instance
convert variable references to function calls
replacing statement list with function call
inlining a function call
change the superclass of a class
moving a member variable to a superclass
moving a member variable to a subclass

Table 2.1: Primitive Refactorings Defined by William Opdyke

inherited from their superclasses.

5. *Compatible Signatures in Member Function Redefinition*—In C++, it is critical that overriding methods have the same signature as the overridden method.

6. *Type-Safe Assignments*—After a refactoring, the left-hand side of every assignment must be of the type or a subtype of the type of the variable on the right-hand side.

7. *Semantically Equivalent References and Operations*— Semantic equivalence was defined operationally. The program had to produce the same value for a given set of inputs both before and after a refactoring.

Formal proofs of the last property are difficult to produce. Additionally, there are some useful transformations that do not preserve program behavior, but alter it in specific ways. To accommodate this, we have changed the definition of refactoring to simply be a program transformation that has a precondition that a program must satisfy for the refactoring to be legally applied. This allows us to both avoid formal proofs of correctness and to allow refactorings that do not preserve program behavior. Opdyke's refactorings still fit into this definition. Additionally, we augment his definition of refactorings with postconditions. This simplifies the construction of composite refactorings and reduces the amount of analysis required to perform later refactorings.

On of the biggest changes in the OO community in recent years has been the introduction and widespread acceptance of the Java programming language [GJS96]. This language removes some of the more problematic constructs from C++ and therefore constructing refactoring tools for the language should be simpler. C++ refactoring-type tools, such as Sniff++, were fairly limited in their capabilities due to the difficult syntax of C++. Another problem that C++ presents is its widespread use of pointers and pointer arithmetic. Java does not allow arbitrary pointer operations, avoiding these difficulties. More recently, members of William Griswold's group have been developing a tool to assist in restructuring Java programs. His approach has been mainly in visualizing Java programs with a star diagram [Kor98]. We will probably focus future research on creating practical, commercial-grade refactoring tools for Java.

2.4 Evolutionary Software Development

One of the ideas that has been introduced in the study of software maintenance is *software entropy*. Software entropy is the observation that over time, the structure of software decays. Fred Brooks says:

> All repairs tend to destroy the structure, to increase the entropy and disorder of the system. Less and less effort is spent on fixing original design flaws; more

> and more is spent on fixing flaws introduced by earlier fixes. As time passes, the system becomes less and less well-ordered. Sooner or later the fixing ceases to gain any ground [Bro75].

In a perfectly-structured system, every change would be completely localized, and therefore, maintenance cost should be approximately constant. However, no system is perfectly structured and as the system grows it will become more unstructured. Whenever a programmer is required to add any function that was not envisioned in the original design, he will attempt to find someplace in the structure of the code to add the feature that will simply work. This will degrade the structure. Likewise, repairs will degrade the structure since they will be made in the simplest and quickest way [BL71]. As the structure decays, the complexity of the software grows, increasing the likelihood that future enhancements will be made in the wrong place.

Changes will continue to occur to a software system throughout its lifetime. This is embodied in Lehman's first law of program evolution:

> A large program that is used undergoes continuing change or becomes progressively less useful. The change process continues until it is judged more cost-effective to replace the system with a recreated version [Leh78].

His second law describe the increase in complexity that occurs with this environment of change.

> As a large program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it.

The idea of *software evolution* has been explored in the area of framework design [JF88, RJ98] and, more recently, as a program development methodology [GHH97][Bec99]. Software evolution is the idea that as a program is used it must change to meet the changing requirements in a way that improves its design rather than degrades it. This idea has several implications.

Parnas described a good modularization, or design, as one that allows anticipated changes to be made within a single module. This is accomplished by hiding the details of the *how* the modules works behind a fixed interface of the *what* a module achieves [Par72]. However, determining what is going to change in the future of a program is extremely difficult. The fundamental problem is that people are notoriously bad guessers when it comes to creating software designs. The tendency is to put in flexibility that never gets used, and to overlook places where the software ends up changing. As a result, the design is more complicated than it needs to be in places, and inadequate in others. This results in a system that is incredibly brittle. Only with a lot of experience with a particular type of software system will a designer create an adequate design.

Extreme Programming (XP) is a methodology developed by Kent Beck whose central idea is that you work on one use-case at a time and only design the software to handle the use-case that you working on. If a particular use-case does not fit well into the design, refactor the design until the use-case can be implemented in a reasonable manner. Rather than trying to avoid change, XP is a methodology that embraces change. One of the key aspects of Extreme Programming is continual and aggressive refactoring. Without refactoring, XP will not work. This approach was successfully used to implement the Chrysler Comprehensive Compensation payroll system [GHH97].

Entropy and evolution both involve the expenditure of energy. The main difference is when the design occurs. In entropic systems, the design is performed up-front and once the coding has commenced, is never changed, or if it is, it is with great pain. An exponentially increasing amount of energy is expended maintaining the system until it finally dies and is replaced. In evolutionary systems, the design is created and improved as the various forces manifest themselves. A constant expenditure of energy is used to evolve the design to adapt to the system's environment. It is my belief that this approach is the right way to develop software, but can only succeed in the presence of automated refactoring tools that allow the programmers to adjust the design at a relatively low cost, thus keeping the constant expenditure of energy low.

2.5 Design Patterns

Software experts have within themselves a knowledge of how to solve various problems that they have encountered. One of the recent trends in software has been to attempt to capture this knowledge in a literary form known as *patterns.* A pattern consists of the solution to a problem in a given context that resolves a set of forces. The context of a pattern establishes the relative importance of the various forces that impinge upon the solution. After the solution is presented, there is typically a discussion of the solution that reveals the consequences of a particular solution. A particular problem can have different solutions within different contexts. The concept of patterns was developed by the architect Christopher Alexander for his particular domain. Alexander says,

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice [AIS+79].

Alexander's ideas resonated with members of the software community and they began to write down patterns that they were familiar with. Several recent books have presented catalogs of patterns [GHJV95, BMR+96, ABW98]. These catalogs of design patterns create many opportunities for refactoring.

Often the design embodied in a design pattern is not realized in the software. This can be due to either the designer's unfamiliarity with the design pattern that is applicable to the problem at hand, or the requirements change to make the design pattern applicable where is wasn't before. In either case, refactoring is necessary to make the old design and code fit the new design. In this view, design patterns correspond to composite refactorings [RBJ97].

These design pattern catalogs have also created the opposite problem. It has become common for someone who reads a book on design pattern to apply those patterns within their software, even when the added flexibility is not needed. In these cases, the software would be more understandable if the design pattern is not present. Undoing this misguided design decision is also a refactoring.

Chapter 3

A Model of Refactoring

Opdyke and Johnson defined refactorings as program transformations that preserve program behavior. There are several problems with this informal definition. First, what is meant by "behavior preserving?" A common definition of behavior preservation for a program is that it must produce the same output for a given input. For example, a program that changes the execution time of a routine by 10 milliseconds, without altering its inputs or outputs, would normally be considered behavior preserving. However, in a hard real-time application, such a transformation is most definitely *not* considered behavior preserving. The second problem is that, even with this definition of behavior preservation, proving this property is impossible for all but the most trivial of program transformations. This is even more difficult in a programming language such as Smalltalk which has state and reflective facilities available to the programmer. If a programmer writes code such as `self instVarAt: 2 put: #value`, which sets the value of the second instance variable, even the simple *add instance variable* refactoring can break the program since it can change the layout of the instance variables within an object. Third, there are some program transformations which are not behavior-preserving, but alter the program's behavior in a very specific manner. For example, a programmer might want to replace every occurrence of an expression with a call to a function that implements a similar, but not identical, function. This type of transformation is often referred to as a refactoring, also.

3.1 Definition of Refactoring

Therefore, in the context of this dissertation, we will define refactorings as program transformations that have particular preconditions that must be satisfied before the transformation can be legally performed. For example, the *remove class* refactoring has the preconditions that the class must exist in the system and be unreferenced. Our definition encompasses both behavior-preserving and non-behavior-preserving transformations. The following definition formalizes this.

Definition 3.1.1 *A **refactoring** is a pair $R = (pre, T)$ where pre is the precondition that the program must satisfy, and T is the program transformation.*

This definition will be extended in the next chapter as we augment it to support computing dependencies between refactorings. This definition does not mention how pre and T are specified. In our work, we specify pre with first order predicate calculus. In an environment where timing is important, pre will probably include time constraints. The remainder of this chapter will describe the refactorings that we have studied and the particular framework we have selected for describing the preconditions of these refactorings.

3.2 Program Transformations

Most programming languages have textual representation for entry and storage. However, this form is particularly poor for expressing many of the relationships between a program's elements. Compiler developers learned this long ago and developed **abstract syntax trees**. Other representations have been developed to capture different information about programs, such as control flow graphs (CFGs) and program dependency graphs (PDGs). However, when a tool uses multiple representations, they must be kept synchronized. This can be computationally expensive, especially for tools that change the code frequently and are intended to be used interactively [Gri91]. One way to avoid this is to use incremental

‘ ‘ @ obj size → ‘ ‘ @ obj mySize
x := ‘ ‘ @ obj → self x: ‘ ‘ @ obj

Figure 3.1: Transformation Rules

update, but this is complex and costly to implement [Gri93]. Morgenthaler found algorithms to incrementally extract the same information contained in CFGs and PDGs from ASTs on demand, eliminating the need to create and maintain these expensive data structure [Mor97].

The most natural way of expressing transformation on programs is with tree-to-tree transformation rules. These rules are expressed as source code pairs with special variables for matching various types of subtrees. When this code is parsed, a pair of pattern trees is produced. The tree matcher can then use this pair to transform the program. The first pattern describes the pattern to be matched, and the second pattern describes how to rewrite the first pattern. Two examples are given in Figure 3.1. The first rule changes all sends of the message size to the message mySize. The second rule replaces all assignments to the variable x with calls to the x: method on self. The rewriter will be discussed in more depth in Section 7.2.4.

This dissertation examines several refactorings. One class of refactorings that we do not consider are refactorings that take place mainly below the method level such as replacing an expression with a temporary initialized to that expression, or extracting a portion of a method into a new function. William Griswold examined these transformations, and his most of his work is directly applicable to the problem [Gri91]. The next sections describe the refactorings that we examined.

3.2.1 Class Refactorings

These refactorings change the relationships between the classes in the system.

AddClass($className, superclass, subclasses$) Adds a new, unreferenced class to the system. It can be used to insert a class into the middle of the inheritance hierarchy.

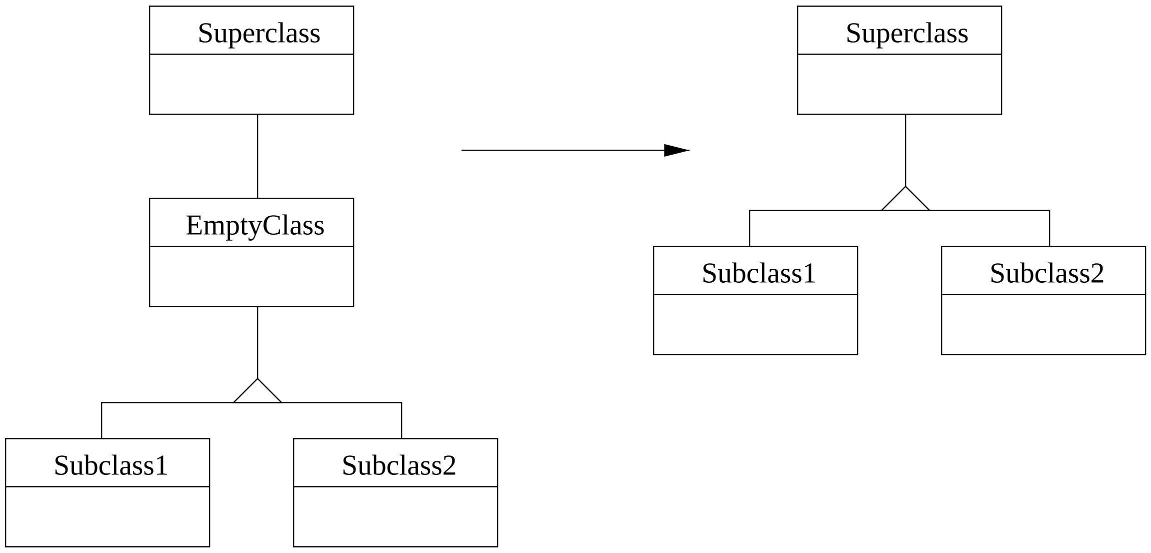


Figure 3.2: *Remove Class* Refactoring with an Empty Class

Therefore, you must specify not only its new superclass, but also the set of classes that were subclasses of the original superclass that are now its subclasses.

RenameClass$(class, newName)$ Renames a class and updates all references to it. No class or global named *newName* can exist for this to be a legal refactoring.

RemoveClass$(class)$ Removes a class from the system. There are two cases where this is legal.

1. If the class is unreferenced and has no subclasses.
2. If the class is unreferenced, has subclasses, but has no methods or instance variables. In this case, the class is removed and all of its subclasses become subclasses of the original class's superclass. Figure 3.2 shows case 2.

3.2.2 Method Refactorings

These refactorings change the methods within the system. Most of them are analogous to operations that can be performed in most non-object-oriented languages. The added complexity that object-oriented languages brings to these is encapsulation and polymorphism.

AddMethod($class, selector, body$) Adds a new, unreferenced method to the system. This is legal as long as *class* and all superclasses of *class* do not already define a method with the same name. Methods with the same name can be added to a subclass as long as it is semantically equivalent to the method defined in the superclass. By adding an equivalent method to a subclass and removing the method from the superclass, we have effectively pushed the method down into the subclass.

RenameMethod($classes, selector, newName$) Changes the name of a set of existing methods and updates all calls to them. If the method being renamed is used polymorphically in any of the classes in the set, all the classes that are polymorphic with respect to the method being renamed must also be in the set. It would not be a refactoring to rename a single method that is used polymorphically with another. For example, to use this refactoring to rename the add: method on any of the Collection classes, you would have to include all of the subclasses of Collection, since they can be used polymorphically with respect to this method.

RemoveMethod($class, selector$) Removes an unreferenced method from the system. This refactoring is also legal if the method is semantically equivalent to the method with the same name implemented in some superclass.

MoveMethod($class, selector, destVarName, newSelector$) Move this method from its defining class to the class of *destVarName* and give it the name *newSelector*. This move is fairly complex. The original method is replaced with a forwarding method that simply calls *newSelector* on *destVarName*. To allow the moved method to refer to the members of its original class, *newSelector* must take an additional parameter that contains the original object.

3.2.3 Variable Refactorings

AddInstanceVariable($class, varName, initClass$) Adds a new, unreferenced instance variable to *class* which is initialized to *initClass*.

RemoveInstanceVariable($class, varName$) Remove an instance variable from *class*. This is only legal if there are no references to the variable in *class* or any of its subclasses.

RenameInstanceVariable($class, varName, newName$) Rename an instance variable and update all references to it.

PullUpInstanceVariable($class, varName$) Removes an instance variable from all subclasses that define it and adds it to *class*. This can add an additional instance variable to classes that did not define it. This is still a refactoring, though, since the added variable is unreferenced.

PushDownInstanceVariable($class, varName$) Remove an instance variable from *class*. And adds it the immediate subclasses of *class*. This is only legal if there are no references to the variable in *class*.

AbstractInstanceVariable($class, varName, getter, setter$) Replace all direct references to an instance variable with calls to accessor functions. All reads of the variable are replaced with a call to *getter* and all writes of the variable are replaced with a call to *setter*.

MoveInstanceVariable($class, varName, destVarName$) Remove an instance variable and add it to the class of the object stored in *destVarName*. This is only legal if the relationship between all instances of *class* and the instances stored in *destVarName* is always one-to-one. In this case, *destVarName* is known as an **exclusive component**.

3.3 Analysis Functions

Transforming programs is easy. Preserving behavior is hard. To ensure that a particular refactoring is legal, the program must meet certain criteria. A program must be analyzed to determine if it meets these criteria.

The analysis functions must describe the relationships between program components. In an object-oriented language, they must describe the relationships between methods, classes, and instance variables. Some of these relationships, like *instance variables of a class*, can be easily determined from the static program text. Others, like *is exclusive component*, are impossible to compute. The difficulty of computing some functions is highly dependent on the particular language used. For instance, in Smalltalk, type information is often difficult to derive from static text, while in Java, it is trivial.

The analysis functions used in the refactorings can be divided into two categories, primitive and derived. In the definitions of the refactorings given in Appendix A, the preconditions are given in terms of both types of analysis functions. The postconditions are given only in terms of the primitive analysis functions. The next two sections describe each function briefly and, for the derived functions, gives the corresponding definition.

3.3.1 Primitive Analysis Functions

This section describes the primitive functions that form the basis for the program analysis that ensures valid refactorings. These are the fundamental functions that a program analysis framework must compute to implement the refactorings this dissertation examines.

IsClass(*class*) True if a class named *class* exists in the system, false otherwise.

IsGlobal(*name*) True if a global variable named *name* exists in the system, false otherwise.

ClassReferences(*class*) The set of all class, selector pairs of all methods that directly reference *class*.

Superclass($class$) The immediate superclass of *class*. If there is no immediate class, returns $\perp$.

Method($class, selector$) The abstract syntax tree representing the method defined by *selector* in *class*.

ReferencesToInstanceVariable($class, varName$) The set of class, selector pairs of all of the methods that refer to the instance variable name *varName* defined in class *class*.

Senders($class, selector$) The set of all call nodes that call the method defined by *selector* in *class*.

ClassOf($class, varName$) The set of classes of the values that are stored in the instance variable *varName* of *class*.

InstanceVariablesDefinedBy($class$) The set of instance variables defined by *class*.

IsExclusive($class, varName$) True if the object stored in instance variable *varName* is only referenced by a single instance of *class* at a time.

3.3.2 Derived Analysis Functions

This section defines the derived analysis functions. These functions are useful to describe the preconditions of refactorings, but can be computed from the primitive analysis functions.

Subclasses($class$) The set of all immediate subclasses of *class*.

$$\text{Subclasses}(class) \equiv \{c | \text{Superclass}(c) = class\}$$

Subclasses*($class$) All of the subclasses of *class*.

$$\text{Subclasses}^*(class) \equiv \begin{cases} \emptyset & \text{if Subclasses}(class) = \phi \\ \bigcup_{c \in \text{Subclasses}(class)} \{c\} \cup \text{Subclasses}^*(c) & \text{otherwise} \end{cases}$$

IsEmptyClass$(class)$ True if the class has no methods and no instance variables defined on it.

$$\begin{aligned}\text{IsEmptyClass}(class) \equiv \neg\exists s.\text{DefinesSelector}(class, s)\\ \wedge\neg\exists v.\text{DefinesInstanceVariable}(class, v)\end{aligned}$$

Superclass$^*(class)$ All of the ancestors of *class* in the inheritance hierarchy.

$$\text{Superclass}^*(class) \equiv \begin{cases} \emptyset & \text{if Superclass}(class) = \perp \\ \{\text{Superclass}(class)\} \cup \text{Superclass}^*(\text{Superclass}(class)) & \text{otherwise} \end{cases}$$

DefinesSelector$(class, selector)$ True if *selector* is defined in *class*.

$$\text{DefinesSelector}(class, selector) \equiv \text{Method}(class, selector) \neq \perp$$

LookedUpMethod$(class, selector)$ Returns the body of the method that will get called if *selector* is sent to *class*.

$$\text{LookedUpMethod}(class, selector) \equiv \begin{cases} \perp & \text{if } class = \perp \\ \text{Method}(class, selector) & \text{if Method}(class, selector) \neq \perp \\ \text{LookedUpMethod}(\text{Superclass}(class), selector) & \text{otherwise} \end{cases}$$

UnderstandsSelector$(class, selector)$ True if an instance of *class* responds to *selector*.

$$\begin{aligned}&\text{UnderstandsSelector}(class, selector) \equiv\\ &\exists c \in \{class\} \cup \text{Superclass}^*(class).\text{DefinesSelector}(c, selector)\end{aligned}$$

DefinesInstanceVariable$(class, varName)$ True if *class* defines an instance variable named *varName*.

$$\text{DefinesInstanceVariable}(class, varName) \equiv varName \in \text{InstanceVariablesDefinedBy}(class)$$

HierarchyDefinesInstVar$(class, varName)$ True if any subclass or any superclass defines an instance variable named *varName*.

$$\text{HierarchyDefinesInstVar} \equiv \exists c \in \{class\} \cup \text{Superclass}^*(class) \cup \text{Subclasses}^*(class). \\ \text{DefinesInstanceVariable}(c, varName)$$

HierarchyReferencesInstVar$(class, varName)$ True if any subclass or any superclass references an instance variable named *varName*.

$$\text{HierarchyReferencesInstVar}(class, varName) \equiv \\ \exists c \in (\{class\} \cup \text{Superclass}^*(class) \cup \text{Subclasses}^*(class)). \\ \text{ReferencesToInstanceVariable}(c, varName) \neq \emptyset$$

3.4 Abstract Syntax Tree Functions

Many of the refactorings require transforming abstract syntax trees within the program. These transformations impact the results of future analysis functions. This section describes briefly the abstract syntax tree transformation functions that are necessary for eliminating analysis.

IsGetter$(method, variable)$ Returns true if *method* is a getter function for *variable*. A getter is a method that simply returns the value of the variable.

IsSetter$(method, variable)$ Returns true if *method* is a setter function for *variable*. A setter function takes one argument and assigns it to the variable that it sets.

AbstractVariable$(method, variable, getter, setter)$ Transforms *method* so all direct reads of *variable* become calls to *getter* and all direct writes become calls to *setter*.

MovedMethod$(method)$ Transforms *method* so all *self* become references to an additional parameter.

ForwarderMethod$(selector, destVar)$ Returns a method that simply forwards *selector* to variable $destVar$ and passes *self* as an additional argument.

UnboundVariables$(method)$ Returns the set of variables that are used by *method* but not defined as temporaries.

RenameReferences$(method, oldName, newName)$ Returns the method with all references to oldName renamed to newName.

3.5 First Order Predicate Logic

In our work, we specify the refactorings' preconditions in first-order predicate logic (FOPL). In addition to the standard logic symbols $\{\neg, \wedge, \vee, \rightarrow, \equiv, =, \exists, \forall\}$, FOPL includes a set of extralogical symbol that specify the various functions and predicates. A *basis* for predicate logic is a pair $B = (F, P)$ where F is the set of function symbols and P is a set of predicate symbols. The functions and predicates that we have described in the previous sections make up the basis that we use. Since a well-formed formula in FOPL has no meaning apart from an interpretation, we must also define what interpretation we will be using to assign meanings to well-formed formulae. An interpretation of FOPL, $\mathcal{I} = (D, \mathcal{I}_0)$, is a pair where D is the domain of discourse and $\mathcal{I}_0$ is a mapping which assigns values from within D to the symbols within F and P [LS87].

In our application, the domain D consists of program elements. In particular, it contains the classes, variables, and methods of the program that is being refactored. The predicate and function symbols represent various program analyses that can be computed for the program.

Theoretically, these are implicitly defined by the program, but practically, we must compute them. Therefore, each program induces a specific interpretation of FOPL. We designate the interpretation that a program P induces as $\mathcal{I}_P$. To determine the truth value of a precondition given in FOPL, we must evaluate it with respect to an interpretation. The evaluation of a precondition *pre* with respect to an interpretation $\mathcal{I}_P$ is denoted by $\models_{\mathcal{I}_P} pre$.

Now that we have given the formalisms that we will use to define the refactorings in this dissertation, we specify exactly what a legal refactoring is. Informally, a legal refactoring is one whose preconditions are satisfied by the program. Formally,

Definition 3.5.1 *A refactoring, $R = (pre, T)$, is **legal** for a program P iff $\models_{\mathcal{I}_P} pre$.*

Chapter 4

Dependencies Between Refactorings

The refactorings specified both in this dissertation and Bill Opdyke's dissertation are very simple. However, small refactorings, if they preserve behavior, can be composed into larger behavior-preserving refactorings, with earlier refactorings changing the program so it meets the prerequisites of later refactorings. If one refactoring "sets up" the preconditions for a second refactoring, the latter refactoring is said to be *dependent* upon the first. We will formalize this relationship in this chapter and show how to compute this relationship between several types of refactorings.

Atomic refactorings are rarely performed in isolation. They are too small and accomplish too little individually. They are usually part of a group of refactorings intended to cause a design change. There are interesting properties that can be computed from this group. Therefore, this collection of refactorings applied sequentially needs to be formalized.

Definition 4.0.2 *A* ***chain*** *of refactorings is a sequence of legal refactorings,* $< R_1, R_2, \ldots, R_n >$*, that are applied to a program sequentially with no intervening changes to the program.*

Since each refactoring is legal, the program must satisfy the preconditions of each refactoring when it is applied.

4.1 Eliminating Analysis with Postconditions

Given the context in which most refactorings are performed, namely within chains, we can reduce the amount of program analysis that must be performed if we can formally specify how each refactoring affects *all* of the analysis functions that are used in evaluating the preconditions of the refactorings. Analysis functions are typically computed lazily. A program analysis framework rarely computes the *entire* function, but only those values that are actually requested from it. For instance, it will not compute the function Senders($class$, $selector$) for all classes and selectors, but only those classes and selectors that affect a particular refactoring. So, by incrementally updating these functions as the refactorings are applied, we can avoid analyzing the program to find values for analysis functions that were asserted by earlier refactorings.

Therefore, we will augment the definition of a refactoring from the previous chapter with postconditions. The new definition includes a third term that specifies how the interpretation of the assertions changes as a result of this refactoring. We will refer to this third term as the **postconditions** of the refactoring, although it is a transformation on assertions and not an assertion itself. By doing this, we have moved the domain of discourse of the refactoring from programs to the language used to specify the preconditions. This language is typically easier to reason about than programs.

Definition 4.1.1 *A* ***refactoring*** *is an ordered triple $R = (pre, T, P)$ where pre is an assertion that must be true on a program for R to be legal, T is the program transformation, and P is a function from assertions to assertions that transforms legal assertions whenever T transforms programs.*

This abstract definition does not specify how the preconditions or the assertions are represented. There are several ways that the components could be specified. In our research, we use relations between program elements to specify the preconditions and assertions. In a real-time environment, the preconditions would include timing constraints and the postconditions would have to specify how those timings changed due to the program transformation.

This dissertation uses first order predicate calculus to specify the precondition assertions for the refactorings. Therefore, we specify the postconditions of the refactorings as functions from interpretations of FOPC to interpretations. Appendix A presents the full definition of all of the refactorings that we have studied. These definitions let us show that a refactoring within a chain is legal with less program analysis by using the postconditions of the refactorings earlier in the chain to verify its preconditions.

One of the common uses for refactorings is to alter a design to incorporate a design pattern [GHJV95]. For example, the *Strategy* pattern encapsulates algorithms within a family of objects rather than in the object that is its client. An object that represents a composition of text may have strategy objects to represent how it is to be rendered onto different devices. Usually, this is not the original design. The first approach is to simply add the algorithms to the composition object. At some point the designer realizes that a better approach is to refactor the code to use the Strategy pattern. To do this, he creates a new class to represent a particular algorithm, adds a new instance variable to reference the newly created class, and moves the methods that implement the algorithm into the new class, along with any instance variables that are specific to the algorithm. Using the notation for refactorings that we established earlier, the steps are:

1. AddClass$(StrategyClass, Object, \phi)$
2. AddInstanceVariable$(Monolith, currentStrategy, StrategyClass)$
3. MoveInstanceVariable$(Monolith, foo, currentStrategy)$
4. MoveMethod$(class, methodName, currentStrategy)$

Of course, steps 3 and 4 can be repeated as many times as necessary to move multiple variables or methods into the new class. In steps 3 and 4, performing the refactoring requires determining the class of *currentStrategy*. In general, this is difficult to do well in Smalltalk. However, using postconditions of steps 1 and 2, we can assert that the instance variable is of class *StrategyClass*. Moreover, step 3 requires that the two classes always exist in a 1-to-1 relationship. This is extremely difficult to prove in any language. But again, the postconditions of steps 1 and 2 allow us to assert that fact.

4.2 Deriving Preconditions of Composite Refactorings

We have already expressed the value of composite refactorings. Creating composite refactorings to be used by a programmer presents a problem with preconditions. Each atomic refactoring within the composite has its own preconditions that must be true for the refactoring to be legal. Given a sequence of refactorings that make up a composite refactoring, it would be nice to be able to derive the preconditions for the composite refactoring from the individual refactorings that it comprises.

The solution to this problem is not as simple as ANDing all of the preconditions from the atomic refactorings. This is because the results of earlier refactorings will satisfy some of the later preconditions. In fact, the later preconditions will not all be true at the start. The preconditions of the first refactoring in the composite must be true, but later ones may be implied by earlier refactorings. In the example from the previous section, the preconditions for step 2 are not true in the initial program since the class of the variable being added does not exist until after step 1.

By using the information provided by the postcondition assertions, we can derive the composite refactoring's preconditions. We do this by evaluating the preconditions of each refactoring in the interpretation that has been transformed by earlier refactorings in the composite. Either the transformed environment allows us to evaluate the precondition, or it has the same value as it would have had in the initial interpretation.

Given a chain of refactorings, $< R_1, R_2, R_3, \ldots, R_n >$, that is legal on program P, we know the preconditions of R_1 must be true intially, the preconditions of R_2 must be true after the postconditions of R_1 have been applied, and so on. Mathematically:

$$\models_{\mathcal{I}_P} pre_1 \wedge \models_{P_1(\mathcal{I}_P)} pre_2 \wedge \models_{P_2(P_1(\mathcal{I}_P))} pre_3 \wedge \ldots \wedge \models_{P_{n-1}(\ldots P_2(P_1(\mathcal{I}_P))\ldots)} pre_n$$

By using this observation together with the formal definitions, we can derive the preconditions for a composite refactoring. Consider the refactoring that creates a Strategy object. The refactoring is:

$$\begin{aligned}
&< \text{AddClass}(Object, Strategy, \phi),\\
&\text{AddInstanceVariable}(Monolith, currentStrategy, Strategy),\\
&\text{MoveMethod}(Monolith, implementor, currentStrategy, newImplementor),\\
&\text{MoveInstanceVariable}(Monolith, usedByImplementor, currentStrategy) >
\end{aligned}$$

Obviously, $pre_{AddClass}$ will be part of the preconditions for the entire composite. For each of the other refactorings, we will derive which of the precondition clauses are satisfied, and which ones are part of the precondition for the entire composite refactoring.

First, we rewrite the preconditions for the *add instance variable* refactoring with the postconditions of the *add class* refactoring.

$$\begin{aligned}
&\models_{P_{AddClass}(\mathcal{I}_P)} pre_{AddInstanceVariable} =\\
&\quad \text{IsClass}[Strategy/true](Monolith)\\
&\quad \wedge\text{IsClass}[Strategy/true](Strategy)\\
&\quad \wedge\neg\text{HierarchyDefinesInstVar}(Monolith, currentStrategy)\\
&\quad \wedge\neg\text{IsGlobal}(currentStrategy)
\end{aligned}$$

The IsClass$[Strategy/true](Monolith)$ clause is the same as IsClass$(Monolith)$ since the argument to IsClass is not *Strategy*. The IsClass$[Strategy/true](Strategy)$ clause simplifies to *true* since we know the value of the IsClass predicate at that point. The other two clauses are not affected by the postconditions of the *add class* refactoring. Therefore, the contribution of the *add instance variable* refactoring to the preconditions of composite refactoring is:

$$\begin{aligned}
&\text{IsClass}(Monolith)\\
&\wedge\neg\text{HierarchyDefinesInstVar}(Monolith, currentStrategy)\\
&\wedge\neg\text{IsGlobal}(currentStrategy)
\end{aligned}$$

We then derive the contributions of the remaining refactorings in the composite.
Let $\mathcal{I}' = P_{AddClass}(\mathcal{I}_P)$.

$$\begin{aligned}
&\models_{P_{AddInstanceVariable}(\mathcal{I}')} pre_{MoveMethod} =\\
&\quad \text{IsClass}[Strategy/true](Monolith)\\
&\quad \wedge\text{DefinesSelector}(Monolith, implementor)\\
&\quad \wedge\forall c. \in \text{ClassOf}[(Monolith, currentStrategy)/\{Strategy\}]\\
&\qquad (Monolith, currentStrategy).\neg\text{UnderstandsSelector}(c, newImplementor)\\
&= \ \text{IsClass}(Monolith)\\
&\quad \wedge\text{DefinesSelector}(Object, implementor)\\
&\quad \wedge\neg\text{UnderstandsSelector}(Strategy, newImplementor)
\end{aligned}$$

Let $\mathcal{I}'' = P_{AddInstanceVariable}(\mathcal{I}')$.

$$
\begin{aligned}
&\models_{P_{MoveMethod}(\mathcal{I}'')}\ pre_{MoveInstanceVariable} = \\
&\qquad \text{IsClass}[Strategy/true](Monolith) \\
&\qquad \wedge \text{DefinesInstanceVariable}[(Monolith, currentStrategy)/true](Monolith, usedByImplementor) \\
&\qquad \wedge \text{DefinesInstanceVariable}[(Monolith, currentStrategy)/true](Monolith, currentStrategy) \\
&\qquad \wedge \forall c \in \text{ClassOf}[(Monolith, currentStrategy)/\{Strategy\}](currentStrategy). \\
&\qquad\qquad \neg\text{HierarchyDefinesInstVar}(c, usedByImplementor) \\
&\qquad \wedge \text{IsExclusive}[(Monolith, currentStrategy)/true](Monolith, currentStrategy) \\
&\quad = \ \text{IsClass}(Monolith) \\
&\qquad \wedge \text{DefinesInstanceVariable}(Monolith, usedByImplementor) \\
&\qquad \wedge \neg\text{HierarchyDefinesInstVar}(Object, usedByImplementor)
\end{aligned}
$$

By combining all of the above clauses and eliminating duplicates, we arrive at the complete preconditions for the composite refactoring.

$$
\begin{aligned}
& \text{IsClass}(Object) \\
\wedge\ & \neg\text{IsClass}(Strategy) \\
\wedge\ & \neg\text{IsGlobal}(Strategy) \\
\wedge\ & \text{IsClass}(Monolith) \\
\wedge\ & \neg\text{HierarchyDefinesInstVar}(Monolith, currentStrategy) \\
\wedge\ & \neg\text{IsGlobal}(currentStrategy) \\
\wedge\ & \text{DefinesSelector}(Monolith, implementor)
\end{aligned}
$$

$$\begin{aligned}
&\wedge \quad \neg\text{UnderstandsSelector}(Object, newImplementor)\\
&\wedge \quad \text{DefinesInstanceVariable}(Monolith, usedByImplementor)\\
&\wedge \quad \neg\text{DefinesInstanceVariable}(Object, usedByImplementor)
\end{aligned}$$

4.3 Commutativity Conditions

Refactorings are typically applied in sequence. This allows large design changes to be composed of a sequence of smaller, more primitive refactorings. Since each step is a refactoring, and therefore, behavior-preserving, the entire composition is also a refactoring. For example, converting an inheritance relationship between two objects into a composition relationship can be broken into the following steps:[Opd92]

1. Abstract all instance variable references.
2. Add an instance variable that holds an instance of the new class
3. Add methods that forward unimplemented messages to the new instance variable.
4. Change the superclass of the original object.

Each step of this refactoring is itself a refactoring. Since each step is behavior preserving, the entire composition is behavior-preserving. This composition property is very powerful in that it allows us to define and automate simpler, low-level refactorings, and then compose them into larger, complex refactorings.

Although the refactorings were initially specified in a sequence, they do not necessarily have to be performed in that sequence. In the example above, the first step does not have to come first. However, the step that adds methods that forward to the new instance variable must come after the step that adds the instance variable. Determining which refactorings must occur before other refactorings is an interesting and useful property. Under what conditions will two refactorings commute?

Informally, two refactorings cannot commute if the first refactoring "sets up" the preconditions of the second refactoring. The following definition formalizes this property.

Definition 4.3.1 *Two refactorings, R_1 and R_2* ***commute*** *for a program iff $< R_1, R_2 >$ is a chain and $< R_2, R_1 >$ is a chain and the analysis functions induced by the program are the same after applying either chain.*

This is not the only definition of commutativity. Another definition would be that two refactoring commute if for *every* program that $< R_1, R_2 >$ is a chain, $< R_2, R_1 >$ is a chain. There are refactorings that commute under our definition but not under this definition. For example, consider the following two hypothetical refactorings.

$$\begin{aligned}
R_1 \quad &: \quad \mathit{pre} \equiv \text{IsClass}(A) \wedge (\text{Method}(A, \mathit{someSel}) = \perp) \\
&\quad\; \mathit{post} \equiv \text{Method}' = \begin{cases} \text{Method}[(A, \mathit{someSel})/ < \mathit{MethodBody} >] & \text{if} B \in \text{Superclass}^*(A) \\ \text{Method}[(A, \mathit{someSel})/ \perp] & \text{otherwise} \end{cases} \\
R_2 \quad &: \quad \mathit{pre} \equiv \text{Method}(A, \mathit{someSel}) \neq \perp \\
&\quad\; \mathit{post} \equiv \ldots
\end{aligned}$$

These two refactorings do not commute for every program for which $< R_1, R_2 >$ is a chain. But there are definitely programs for which they will commute, namely those where B is some superclass of A.

The two definitions are the same if the postconditions do not change based on the state of the assertions. Consider a chain $< R_a, R_b >$. Either some of R_b's preconditions are satisfied by the postconditions of R_a, in which case they will not commute, or all of its preconditions were true before the application of R_a and R_a's postconditions did not change them. So, if the chain $< R_a, R_b >$ commutes for some program P_1, but not for a program

P_2, it implies that R_b's preconditions must have been satisfied before R_a in P_1, but satisfied by the postconditions of R_a in P_2. This means that the postconditions of R_a must depend on the state of the assertions. If the postconditions of a refactoring do not depend on the state of the assertions, then if the chain $< R_a, R_b >$ commutes for some program, it must commute for all programs for which it is a legal chain.

In fact, refactorings like R_1 from the example can be rewritten as two refactorings whose postconditions do not depend on the assertions by moving the condition into the precondition. Therefore, R_1 would become:

$$\begin{aligned}
R_{1a} \ &: \ \mathit{pre} \equiv \text{IsClass}(A) \wedge (\text{Method}(A, \mathit{someSel}) = \perp) \\
&\qquad \wedge B \in \text{Superclass}^*(A) \\
&\quad \mathit{post} \equiv \text{Method}' = \text{Method}[(A, \mathit{someSel})/ < \mathit{MethodBody} >] \\
R_{1b} \ &: \ \mathit{pre} \equiv \text{IsClass}(A) \wedge (\text{Method}(A, \mathit{someSel}) = \perp) \\
&\qquad \wedge B \notin \text{Superclass}^*(A) \\
&\quad \mathit{post} \equiv \text{Method}' = \text{Method}[(A, \mathit{someSel})/ \perp]
\end{aligned}$$

This dissertation will use the definition of commutativity given in Definition 4.3.1. From this definition, we derive the following formula for the conditions under which two refactorings commute.

Formula 4.3.1 *Two refactorings, $R_1 = (pre_1, T_1, P_1)$ and $R_2 = (pre_2, T_2, P_2)$ commute on a program Q iff:*

$$\models_{\mathcal{I}_Q} pre_1 \wedge \models_{(P_1(\mathcal{I}_Q))} pre_2 \wedge \models_{\mathcal{I}_Q} pre_2 \wedge \models_{(P_2(\mathcal{I}_Q))} pre_1 \wedge P_1(P_2(\mathcal{I}_Q)) = P_2(P_1(\mathcal{I}_Q))$$

Often, whether two refactorings can commute, depends on the relationship between the arguments of the refactorings. Formula 4.3.1 lets us derive exactly the conditions that must exist between the arguments of two refactorings so that they can commute. In the next two sections, we derive the commutativity conditions between some sample refactorings.

4.4 Example Derivation

To demonstrate how to derive commutativity conditions, consider the two refactorings AddClass($class_1, superclass_1, subclasses_1$) and AddInstanceVariable($class_2, varName_2, initclass_2$).

Given the chain $<$ AddClass($superclass_1, class_1, subclasses_1$), AddInstanceVariable($class_2, varName_2, initClass_2$) $>$, the preconditions for each are as follows:

$$\begin{aligned}\models_{\mathcal{I}} pre_1 \quad : \quad & \text{IsClass}(superclass_1) \\ & \wedge\neg\text{IsClass}(class_1) \\ & \wedge\neg\text{IsGlobal}(class_1) \\ & \wedge\forall c. \in subclasses_1.(\text{IsClass}(c) \wedge \text{Superclass}(c) = superclass_1)\end{aligned}$$

$$\begin{aligned}\models_{P_1(\mathcal{I})} pre_2 \quad : \quad & \text{IsClass}[class_1/true](class_2) \\ & \wedge\text{IsClass}[class_1/true](initClass_2) \\ & \wedge\neg\text{HierarchyDefinesInstVar}(class_2, varName_2) \\ & \wedge\neg\text{IsGlobal}(varName_2)\end{aligned}$$

The commuted chain, $<$ AddInstanceVariable($class_2, varName_2, initClass_2$), AddClass($superclass_1, class_1, subclasses_1$) $>$ has the preconditions:

$$\begin{aligned}
\models_{\mathcal{I}} pre_2 \quad : \quad & \text{IsClass}(class_2) \\
& \wedge \text{IsClass}(initClass_2) \\
& \wedge \neg \text{HierarchyDefinesInstVar}(class_2, varName_2) \\
& \wedge \neg \text{IsGlobal}(varName_2)
\end{aligned}$$

$$\begin{aligned}
\models_{P_1(\mathcal{I})} pre_1 \quad : \quad & \text{IsClass}(superclass_1) \\
& \wedge \neg \text{IsClass}(class_1) \\
& \wedge \neg \text{IsGlobal}(class_1) \\
& \wedge \forall c. \in subclasses_1.(\text{IsClass}(c) \wedge \text{Superclass}(c) = superclass_1)
\end{aligned}$$

These formulae must satisfy $\models_{\mathcal{I}_Q} pre_1 \wedge \models_{(P_1(\mathcal{I}_Q))} pre_2 \wedge \models_{\mathcal{I}_Q} pre_2 \wedge \models_{(P_2(\mathcal{I}_Q))} pre_1$. The only potential conflict arises when the same analysis function appears in multiple clauses with different parameters. In this example, the only conflict arises from the IsClass function. All of the following must be true for these two to commute:

$$\begin{aligned}
& \neg \text{IsClass}(class_1) \\
\wedge \quad & \text{IsClass}[class_1/true](class_2) \\
\wedge \quad & \text{IsClass}[class_1/true](initClass_2) \\
\wedge \quad & \text{IsClass}(class_2) \\
\wedge \quad & \text{IsClass}(initClass_2)
\end{aligned}$$

This can only be satisfied if $class_1 \neq class_2 \wedge class_1 \neq initClass_2$.

The final clause of the formula, $P_1(P_2(\mathcal{I}_Q)) = P_2(P_1(\mathcal{I}_Q))$, can only be violated if the postconditions from the two refactoring affect the same analysis function. With these two refactorings, only the InstanceVariablesDefinedBy function is affected by both. Therefore,

$$\begin{aligned}
&\text{InstanceVariablesDefinedBy}[class_1/\phi] \\
&\quad [class_2/\text{InstanceVariablesDefinedBy}(class_2) \cup \{varName_2\}] = \\
&\text{InstanceVariablesDefinedBy}[class_2/\text{InstanceVariablesDefinedBy}(class_2) \cup \{varName_2\}] \\
&\quad [class_1/\phi]
\end{aligned}$$

This is true if $class_1 \neq class_2$. This condition is already present in the previous expression, therefore, the complete conditions that must be met so these two refactorings commute are:

$$class_1 \neq class_2 \wedge class_1 \neq initClass_2$$

4.5 Another Example

This derivation determines the commutativity conditions between RemoveClass($class_1$) and RemoveClass($class_2$). The preconditions for each are (substituting the primitive definition of the Subclasses function):

$$\begin{aligned}
\models_{\mathcal{I}} pre_1 \quad : \quad & \text{IsClass}(class_1) \\
& \wedge \text{ClassReferences}(class_1) = \phi \\
& \wedge (\{c | \text{Superclass}(c) = class_1\} = \phi \\
& \quad \vee \text{IsEmptyClass}(class_1))
\end{aligned}$$

$$\begin{aligned}
\models_{P_1(\mathcal{I})} pre_2 \quad : \quad & \text{IsClass}[class_1/false](class_2) \\
& \wedge \text{ClassReferences}_1(class_2) = \phi
\end{aligned}$$

$$\wedge(\{c|\text{Superclass}_1(c) = class_2\} = \phi$$
$$\vee\text{IsEmptyClass}(class_2))$$

where $\text{ClassReferences}_1 = \forall c \neq class_1.\forall s.\text{ClassReferences}[class_1/\perp][c/\text{ClassReferences}(c) - \{(class_1, s)\}]$, and $\text{Superclass}_1 = \forall c \in \text{Subclasses}(class_1).\text{Superclass}[class_1/\perp][c/\text{Superclass}(class_1)]$. These equations can be simplified to:

$$\begin{aligned}
\models_{\mathcal{I}} pre_1 \quad : \quad & \text{IsClass}(class_1) \\
& \wedge\text{ClassReferences}(class_1) = \phi \\
& \wedge(\neg\exists c.\text{Superclass}(c) = class_1 \\
& \quad \vee\text{IsEmptyClass}(class_1))
\end{aligned}$$

$$\begin{aligned}
\models_{P_1(\mathcal{I})} pre_2 \quad : \quad & \text{IsClass}[class_1/false](class_2) \\
& \wedge\text{ClassReferences}_1(class_2) = \phi \\
& \wedge(\neg\exists c.\text{Superclass}_1(c) = class_2 \\
& \quad \vee\text{IsEmptyClass}(class_2))
\end{aligned}$$

The preconditions for the commuted chain $< \text{RemoveClass}(class_2), \text{RemoveClass}(class_1) >$ are:

$$\begin{aligned}
\models_{\mathcal{I}} pre_2 \quad : \quad & \text{IsClass}(class_2) \\
& \wedge\text{ClassReferences}(class_2) = \phi
\end{aligned}$$

$$\begin{aligned} &\wedge(\{c|\text{Superclass}(c) = class_2\} = \phi \\ &\quad\vee\text{IsEmptyClass}(class_2)) \end{aligned}$$

$$\begin{aligned} \models_{P_2(\mathcal{I})} pre_1 \quad : \quad & \text{IsClass}[class_2/false](class_1) \\ & \wedge\text{ClassReferences}_2(class_1) = \phi \\ & \wedge(\{c|\text{Superclass}_2(c) = class_1\} = \phi \\ & \quad\vee\text{IsEmptyClass}(class_1)) \end{aligned}$$

where ClassReferences$_2 = \forall c \neq class_2 \forall s.$ClassReferences$[class_2/\perp][c/$ClassReferences$(c) -$ $\{(class_2, s)\}]$, and Superclass$_2 = \forall c \in$ Subclasses$(class_2)$.Superclass$[class_2/\perp][c/$Superclass$(class_2)]$. Again, simplifying yields:

$$\begin{aligned} \models_{\mathcal{I}} pre_2 \quad : \quad & \text{IsClass}(class_2) \\ & \wedge\text{ClassReferences}(class_2) = \phi \\ & \wedge(\neg\exists c.\text{Superclass}(c) = class_2 \\ & \quad\vee\text{IsEmptyClass}(class_2)) \end{aligned}$$

$$\begin{aligned} \models_{P_2(\mathcal{I})} pre_1 \quad : \quad & \text{IsClass}[class_2/false](class_1) \\ & \wedge\text{ClassReferences}_2(class_1) = \phi \\ & \wedge(\neg\exists c.\text{Superclass}_2(c) = class_1 \\ & \quad\vee\text{IsEmptyClass}(class_1)) \end{aligned}$$

First, rearrange the terms to group similar analysis functions.

$$\begin{aligned}
& \text{IsClass}(class_1) \\
\wedge\ & \text{IsClass}[class_1/false](class_2) \\
\wedge\ & \text{IsClass}(class_2) \\
\wedge\ & \text{IsClass}[class_2/false](class_1) \\
\wedge\ & \text{ClassReferences}(class_1) = \phi \\
\wedge\ & \text{ClassReferences}_1(class_2) = \phi \\
\wedge\ & \text{ClassReferences}(class_2) = \phi \\
\wedge\ & \text{ClassReferences}_2(class_1) = \phi \\
\wedge\ & (\neg\exists c.\text{Superclass}(c) = class_1 \vee \text{IsEmptyClass}(class_1)) \\
\wedge\ & (\neg\exists c.\text{Superclass}_1(c) = class_2 \vee \text{IsEmptyClass}(class_2)) \\
\wedge\ & (\neg\exists c.\text{Superclass}(c) = class_2 \vee \text{IsEmptyClass}(class_2)) \\
\wedge\ & (\neg\exists c.\text{Superclass}_2(c) = class_1 \vee \text{IsEmptyClass}(class_1))
\end{aligned}$$

The only constraint imposed by the IsClass clauses is that $class_1 \neq class_2$. By expanding the ClassReferences clauses, we get:

$$\begin{aligned}
& \text{ClassReferences}(class_1) = \phi \\
\wedge\ & \forall c \neq class_1.\forall s.\text{ClassReferences}[class_1/\perp][c/\text{ClassReferences}(c) - \{(class_1, s)\}](class_2) = \phi \\
\wedge\ & \text{ClassReferences}(class_2) = \phi \\
\wedge\ & \forall c \neq class_2.\forall s.\text{ClassReferences}[class_2/\perp][c/\text{ClassReferences}(c) - \{(class_2, s)\}](class_1) = \phi
\end{aligned}$$

These equations impose the additional constraint that ClassReferences($class_1$) $= \phi \wedge$ ClassReferences($class_2$) $= \phi$. Finally, the last set of clauses becomes (with c renamed in the substitution to d for clarity)

$$\begin{array}{ll} & (\neg\exists c.\text{Superclass}(c) = class_1 \\ & \quad \vee\text{IsEmptyClass}(class_1)) \\ \wedge & (\neg\exists c.(\forall d \in \text{Subclasses}(class_1).\text{Superclass}[class_1/\perp][d/\text{Superclass}(class_1)])(c) = class_2 \\ & \quad \vee\text{IsEmptyClass}(class_2)) \\ \wedge & (\neg\exists c.\text{Superclass}(c) = class_2 \\ & \quad \vee\text{IsEmptyClass}(class_2)) \\ \wedge & (\neg\exists c.(\forall d \in \text{Subclasses}(class_2).\text{Superclass}[class_2/\perp][d/\text{Superclass}(class_2)])(c) = class_1 \\ & \quad \vee\text{IsEmptyClass}(class_1)) \end{array}$$

These impose the constraint that Superclass($class_1$) $\neq class_2 \vee$ IsEmptyClass($class_2$). Therefore, the complete conditions to allow the two refactorings to commute are:

$$\begin{array}{ll} & class_1 \neq class2 \\ \wedge & \text{ClassReferences}(class_1) = \phi \\ \wedge & \text{ClassReferences}(class_2) = \phi \\ \wedge & (\text{Superclass}(class_1) \neq class_2 \vee \text{IsEmptyClass}(class_2)) \end{array}$$

4.6 Calculation of Dependencies

Now that we can determine whether two refactorings commute, we can define what it means for one refactoring to depend on another. The following definition formalizes this relationship.

Definition 4.6.1 *Given a chain of refactorings, $< R_1, R_2 >$, R_2* ***depends*** *on R_1 if R_2 and R_1 do not commute.*

If a refactoring is defined to depend upon itself, then the dependency relation is a partial ordering ($\sqsubseteq$). In this relation, $R_1 \sqsubseteq R_2$, if R_2 is dependent on R_1. This relation creates separate chains of refactorings that can be performed independently. This can be seen by the following construction.

Take one of the top refactorings in the chain. If this is not the last refactoring in the chain, commute it with the refactoring that is applied immediately after it in the chain. This is possible since this is a top refactoring in the dependency relation so any refactorings that occur later in the chain do not depend upon it and can be commuted with it. Repeat this process until the top refactoring is at the end of the chain. Take all of the refactorings that are $\sqsubseteq$ to the last refactoring and repeat this process. Repeat until the last refactorings of the chain are the dependent ones. This subchain can now be removed from the original chain. Since there are no dependencies from the new chain to the old chain, these two chains can be performed in any order. Figure 4.1 demonstrates this process graphically.

4.7 Implementation Using Assertions

This dissertation examines 14 refactorings. To create a tool that uses the dependencies between refactorings to perform useful functions, we would have to identify the conditions under which every pair of refactorings commute. To do this would entail 196 derivations similar to the ones given earlier in the chapter. As shown in the earlier sections, derivation of

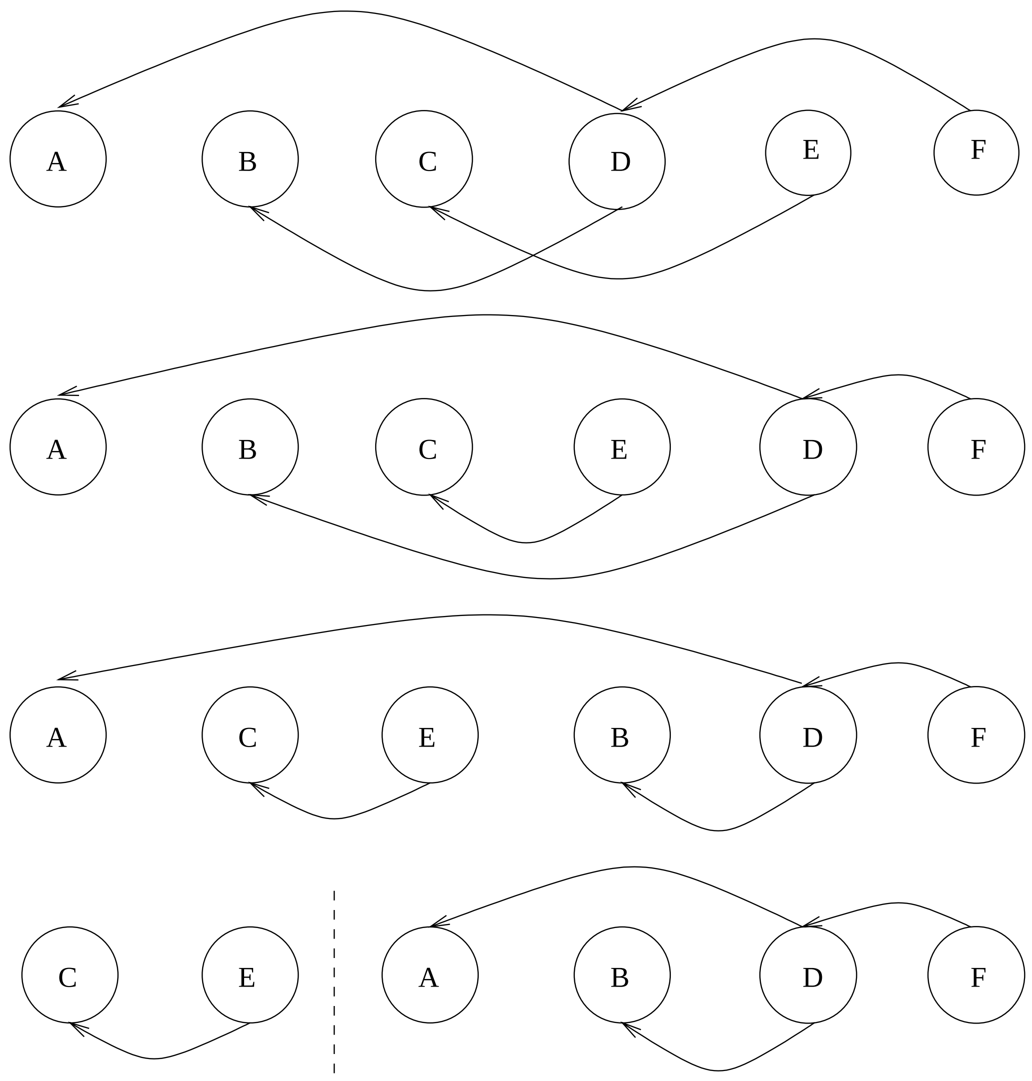


Figure 4.1: Separation of Independent Chains

the commutativity conditions in general can be quite complicated. An alternative approach would be to use a theorem prover to either derive all of the conditions and hard-wire them into a tool, or to use the prover in the tool itself to derive these conditions. Both of these approaches are complicated and time-consuming.

There is a simpler approach we have prototyped for our tool. As each refactoring is performed, its postcondition assertions are added to the database of known facts. If a particular assertion is used to satisfy the preconditions of a later refactoring, then the later refactoring depends on the refactoring that originally asserted the fact. This is a conservative approximation of the dependency relationship since a refactoring *could* assert a condition that was already true. For example, in the chain

$$< \text{AddClass}(Object, Strategy, \phi), \text{AddInstanceVariable}(Monolith, currentStrategy, Strategy) >$$

one of the postconditions of the AddClass refactoring is that $\text{IsClass}' = \text{IsClass}[Strategy/true]$. One of the preconditions of the AddInstanceVariable refactoring is that IsClass(*Strategy*). Since this precondition is satisfied as a result of the assertion from the AddClass refactoring, the AddInstanceVariable refactoring is dependent on the AddClass refactoring. If *Strategy* was already a class before the AddClassasserted it, then this would be a false dependency. This is prevented by the precondition of the AddClassrefactoring that states that the *Strategy* class must not exist for the refactoring to be legal. If refactorings only assert facts that are not true before the application of the refactoring, then this approximation of the dependency relationship is exact.

4.8 Uses Of Dependencies

We can use dependencies between refactorings to implement several types of refactoring tools. This chapter examines three of these applications.

4.8.1 Undo

A refactoring tool should be able to undo a refactoring. Since a refactoring is supposedly behavior-preserving, the inverse operation is itself a refactoring. This inverse, however, may be difficult to implement as a refactoring. For example, consider what happens in the Smalltalk image if a method on an application class is renamed to **add:**. The inverse operation would be rename the method back to its original name. However, determining the senders of the renamed method would be extremely difficult since the Collection hierarchy defines the **add:** method and it is used ubiquitously in the image. Therefore, undo is rarely implemented with inverse refactorings.

One way to implement undo for a refactoring is to simply checkpoint each method it changes and to undo the refactoring by restoring the old version. This approach works, but becomes more difficult whenever chains of refactorings are performed. Undoing a refactoring in the middle of the chain requires undoing some refactorings that were performed later in the chain. For example, if a chain contains the *add class* refactoring followed by an *add method* to that class, undoing the *add class* refactoring will require undoing the *add method* refactoring.

One way to solve this problem is to simply undo all refactorings that were performed later than the target refactoring, undo the target refactoring, and then reapply the later refactorings. If the refactorings are illegal when they are reapplied, discard them. This can be time-consuming for large chains of refactorings.

Dependencies between refactorings can be used to determine which refactorings must be undone. Given an arbitrary chain of refactorings, the dependency DAGs within it can be computed. If a refactoring is undone, all refactorings that occur higher in its dependency DAG, must be undone also.

4.8.2 Parallelizing Refactorings

Another application that arises when applying large numbers of refactorings to a large program is to parallelize the refactoring process. Given a chain of refactorings, you would like to determine which sets of refactorings within the chain can be performed in parallel, and which ones must be performed sequentially.

By using the dependency relationship, we can compute these independent chains. Each chain can be assigned to a separate processor. Even though the refactorings are independent, they may affect the same methods, so some locking mechanisms must be incorporated to prevent two refactorings from corrupting a method. For example, consider two independent renaming refactorings that are updating the senders of the method being renamed. It is quite possible that there is a method within the system that contains calls to both of the methods being renamed. It is obvious that it doesn't matter which order the calls are renamed in, however, while one refactoring is updating the method, it must be locked to prevent the other refactoring from getting a partially refactored method.

4.8.3 Merging Refactorings in Multiuser Programming Environments

Most software is developed by several people. Whenever more than one person is working on a program, there is the potential for conflict. Source code management systems allow coordination between programmers by providing facilities such as version control and merging. Automatic refactoring makes it easy for a single programmer to change a lot of code quickly. However, this makes it much more likely that two people who are refactoring code will generate many conflicts. The dependency relationship can be used by a tool to provide useful information for managing independent developers refactoring a common system.

By using dependency information, a tool can determine if two chains of refactorings that were created by different programmers conflict. This can provide a granularity that is much finer than method-level checking. Two refactorings that don't depend upon each other might affect the same method. With traditional source-code control techniques, someone would have to merge the two versions of every method that both refactorings touched. Since we know that the two refactorings do not depend upon one another, we know that it doesn't matter what order they are performed in. Therefore, the system can be updated by applying both refactorings in any order.

If two sequences of refactorings *do* conflict, it is possible that there are some subsequences from the front of each sequence that do not conflict with each other. Dependency information can be used to find these subsequences so that some of the refactorings can be applied to the system even if the entire sequences cannot.

Chapter 5

Dynamic Refactoring

One way to eliminate expensive static analysis is to defer the analysis until runtime. At runtime, we can observe the program elements that we are interested in and simply record their values. This chapter will look at the advantages and disadvantages of refactoring with this type of analysis.

5.1 Using Runtime Information for Refactoring

The standard and, arguably, preferred method of refactorings code is to statically analyze a program to determine if the preconditions of the refactoring are true. If they are, refactor the code, otherwise, abort the refactoring. In general, every interesting property of programs is undecidable. Therefore, static analyses must use conservative approximations of the actual functions. Usually, this approach is very successful.

The two cases that cause this approach to fail are when the approximation takes a long time to compute and when the approximation is extremely poor. Our refactoring tool is designed to be used interactively, so long delays are not acceptable. If an analysis takes a long time to compute, the programmer will not use a refactoring based on it, but will rather refactor the code by hand, and rely on the test suite to catch any errors. If an approximation is particularly poor, in that it only detects the most degenerate cases, any refactoring based on it will rarely apply. Bill Opdyke's approximation for the IsExclusive analysis falls into this category. His approximation stated that a component was exclusive if it was created and assigned to an instance variable only in the constructor of the object and never assigned anywhere else.

An alternate approach is to use runtime information to perform the analysis. In this approach, the program is refactored, and the code is instrumented to verify that the preconditions of the refactoring hold. If the instrumentation detects that the assumed preconditions are invalid, the refactoring must be undone. To ensure that the preconditions are true, the program must be tested. After the code is sufficiently tested, the instrumentation can be removed.

There are two types of information that are provided by dynamic analysis, predicates and functions. With dynamic predicates, the predicate is assumed to be true unless the instrumentation detects otherwise, at which point it signals its client. For example, the IsExclusive analysis assumes that the component being analyzed *is* exclusively owned until proven otherwise. Predicates are used in the preconditions of refactorings to ensure that they are legal. The other type of analysis is dynamic functions. As the program runs, the analysis provides its client with the values of the function that occur during the execution. For example, the ClassOfanalysis function will return the classes that are assigned to a particular variable. This information is typically used to complete a refactoring. For instance, the *rename method* refactoring must determine which senders of the particular selector being renamed actually refer to the method being renamed. To do this dynamically, the Senders analysis function will provide the call nodes that actually refer to the method at runtime. The refactoring will update these nodes as the program runs.

5.2 Definition of Correctness

Most refactorings are program transformations that are supposed to be behavior-preserving. Therefore, informally, a refactoring is *correct* if the program behaves the same after the transformation as it did before the transformation. However, we have to specify what we mean when we say “behaves the same.” We have to identify the properties of the program that we consider interesting behavior. For example, in most systems a transformation is considered behavior preserving even if it makes the system a millisecond faster or slower. If the same transformation is made on a real-time system, the resulting system can be broken if it violates specified timing constraints.

Therefore, every program is assumed to have a specification. A refactoring is correct if a program that meets its specification continues to meet its specification after the refactoring is applied. Properties of the program that are not in the specification are considered artifacts that will not necessarily be preserved.

5.3 Test Suites as Program Specifications

Program specifications declaratively state the properties that a program should exhibit. Typically, specifications are given to programmers to compare with the program as it is written. A program is correct with respect to a specification if it meets the requirements stated in the specification.

Balzer and Goldman give eight principles of good program specifications:[BG79, Pre87]

1. Separate functionality from implementation.
2. A process-oriented systems specification language is required.
3. A specification must encompass the system of which the software is a component.
4. A specification must encompass the environment in which the system operates.
5. A system specification must be a cognitive model.
6. A specification must be operational.
7. The system specification must be tolerant of incompleteness and augmentable.

8. A specification must be localized and loosely coupled.

Another form of specification of program behavior is the test suite. Test suites satisfy all of the eight principles mentioned above.

1. Test suites are described as input, result pairs without specifying how the result is obtained. In a real-time environment the result will include timing constraints.

2. Pressman defines a process-oriented description as, "the *what* specification is achieved by specifying a model of the desired behavior in terms of functional responses to various stimuli from the environment."[Pre87] Test suites fit this description perfectly.

3. Good test suites verify the interaction between various subsystems within the software. Since these interaction are complex and dynamic, they are also error-prone.

4. Good test suites also account for the interaction of the environment with the software, be it user input or some other form of data acquisition.

5. The cognitive model that a test suite presents is that of a black-box. Anything can happen within the box as long as the outputs for a given set of inputs remains constant. This model meshes very well with the maintenance phase of software development since the primary concern of anyone doing maintenance programming is that they do not introduce new errors into the system.

6. Test suites are the ultimate operational specification. The inputs are given to the program and the results compared against the expected results.

7. Test suites are never complete and yet continue to eliminate many bugs from the system. Whenever users discover bugs in a running system, they augment the test suite by adding tests that will catch the error that slipped through.

8. Individual tests within the test suite do not depend on each other for results. Therefore, as requirements change, tests can be easily added and removed from the test suite.

One of the basic requirements that must be met when any change is made to a system is that the system can continue pass the original test suite. If an extension is made to the system, it must not only pass the original set of tests, but also any additional tests to determine if the extension is correct.

Despite meeting the requirements for a program specification, test suites do have a couple of drawbacks. First, they tend to be quite large. Even though it is practically impossible to test every path through a program, good test suites attempt to exercise each code segment at least once. To achieve this, the program must be executed on large sets of inputs. Test designers attempt to combine as many tests as possible into a single input set. This leads to second problem, namely, duplication. These two forces are opposed to each other. The less duplication, the larger the test suite and the smaller the test suite, the greater the duplication present. Both of these make changing the specification difficult. Another problem with this type of specification is that it is not generative. I would not want to specify a new system solely by its test suite. In the degenerate case, one could write a program that simply looks up the input value and returns the output value.

The definition of correctness that we will be using when discussing dynamic refactoring is based on test suites. A refactoring is *correct* if a program that passes a test suite continues to pass the test suite after the refactoring has been applied. With this definition of correctness, refactorings that require runtime analysis can be shown to be correct over a particular set of executions, namely the test suite.

5.4 Dynamic Analysis vs. Static Analysis

The standard (and arguably, preferred) method of analyzing code is to do it statically. A static analysis examines the source code of the program and computes the appropriate functions. The analysis typically uses a data-flow framework and computes the fixed-point [ASU88]. The main advantage of static analysis is that it doesn't depend on any particular execution of the program, but can provide information about *all* executions of the program.

There are two fundamental shortcomings of the static analysis approach. First, all of the interesting properties of programs are undecidable. That is, they can be reduced to the halting problem [Ric53, HU79]. Therefore, static analyses must compute decidable, conservative approximations of the desired properties. Depending upon the application, a conservative approximation is often satisfactory, but there are several situations where static approximations are inadequate. For example, consider the example of IsExclusive. There are several refactorings that can only be performed if one object is stored in an instance variable of exactly one other object at any given time. Detecting this situation is difficult since detecting if two variables contain the same object at any point in their lifetime is undecidable. The approximation that Opdyke used was to detect degenerate cases that guarantee that a component is exclusive [Opd92]. The most common case that can be detected is that an instance variable is only assigned to a newly created object. This is simple to detect whenever a language contains explicit constructors, such as C++, but in a dynamic language such as Smalltalk without explicit constructors, even this crude approximation defies static analysis.

A solution to the problem is to make a liberal approximation. This can be accomplished by collecting information as the program is executed. The analysis then consists of data that was observed in a running program. Since it is impossible to observe every possible run of the program, the result is still an approximation. However, this approximation errs on the side of incomplete information. For example, a static type analysis may contain more types than those actually assumed by the program element, but a dynamic type analysis may contain fewer types than the actual type of the program element. Until a path through the program is executed, the analyses depending on that path are not computed, therefore, if there are paths that are never executed, they will never be considered by the analyses.

One interesting property of the liberal approximation of dynamic analysis is that when coupling it with conservative, static analysis can sometimes lead to exact information. If the liberal analysis and the static analysis are the same, then the exact value has been determined and no further analysis need be performed.

5.5 Implementation of Dynamic Refactoring

There are two ways to implement the dynamic refactorings discussed in this dissertation, an offline and an online approach. The offline approach instruments a program and collects the information. At some point in the future, this information is used by a tool to transform the source code based on the observed information. The online approach collects the information and transforms the program while it is running. While both of these techniques are reflective, choice of language and environment greatly affects which approach is most feasible. Dynamic environments such as Smalltalk or Lisp lend themselves to the online approach, whereas statically-compiled environments such as C++ or Java are easier to refactor offline. The analysis functions for which we have examined dynamic analyses are Senders, ClassOf, and IsExclusive.

The Refactoring Browser uses a reflective technique known as **method wrappers** to implement the online form of dynamic refactoring [BFJR98]. Method wrappers allow code to be executed before and after the invocation of a method. The analysis is implemented in this before and after code. Another benefit of this technique is that every wrapper in the system has the same code, so recompilation is not necessary to install the wrapper. To install a wrapper on a method, the wrapper is given a reference to the original method, and the wrapped method installed in the method dictionary in place of the original method. The implementation of this technique does not require the recompilation of the method being wrapped. This is an important consideration when instrumenting a large number of methods.

To dynamically compute the senders of a particular method, we put a wrapper on the method we are interested in. At runtime, when the method is called, the wrapper's *before* code is executed. This code examines the call stack to find the class, selector, and call node of the sender and passes this information to its client. After the analysis is complete and any actions are taken by the client, the wrapper passes the execution on to the original method.

Ideally, to analyze the types of variables dynamically, we would use active variables. Every time the value of the variable changed, the analyzer would be notified and the observed type of the variable updated. Our approach is to wrap all methods that write to the variable and determine the class of the variable in the *after* method of the wrapper. This approach has a slight problem. It is possible that within a single method, two writes occur to the variable so it could take on types that we don't see. If the programmer has used accessors everywhere, this will not be a problem, since only the setter function actually writes to the variable, and it will be wrapped. A solution to this problem is to replace all direct writes to the instance variables with calls to a private setter method which we generate and wrap. The current implementation does not do this.

IsExclusive is similar to ClassOf in that a variable is monitored. Since this is a predicate rather than a function, it only notifies its client whenever the conditions is detected to be false. The client assumes that the function is true unless the analysis reports otherwise at runtime. This analysis maintains a table of exclusively-owned objects. If an object is assigned to the variable and is already in the table, the analysis signals its client that the test is false. Otherwise, the old value of the variable is removed from the table and the new value is added. Figure 5.1 show the code that installs the wrappers for the IsExclusive analysis.

5.6 Example of a Dynamic Refactoring

For an example, let's look at performing the MoveInstanceVariable refactoring using dynamic analysis for the IsExclusive and ClassOf analysis functions. One of the preconditions for MoveInstanceVariable($class, varName, destVar$) is IsExclusive($class, destVar$). This is a dynamic predicate. While the program runs, the analysis monitors each assignment to *dest-*

```
IsExclusiveCondition>>install
    | wrapper beforeBlock oldValue instIndex afterBlock env |
    instIndex := class instVarIndexFor: variable.
    beforeBlock := [:rec :args | oldValue := rec instVarAt: instIndex].
    afterBlock :=
                [:rec :args |
                | newValue |
                newValue := rec instVarAt: instIndex.
                owned removeIfPresent: oldValue.
                (owned includes: newValue)
                    ifTrue: [client variable: variable isNotExclusiveIn: class].
                owned add: newValue].
    env := BrowserEnvironment new instVarWritersTo: variable in: class.
    env classesAndSelectorsDo:
                [:cls :sel |
                wrapper := BlockMethodWrapper on: sel inClass: cls.
                wrapper beforeBlock: beforeBlock.
                wrapper afterBlock: afterBlock.
                wrapper install]
```

Figure 5.1: Code to Install Dynamic IsExclusive Analysis

Var and checks to see if any other instance of *class* has that value stored. If the precondition fails, the entire refactoring must be undone using the techniques discussed in the previous chapter. The information that is provided by the ClassOf analysis is used for two purposes at runtime. First, when the analysis reports a class that is stored in the *destVar* and the class has not been seen yet, the precondition $\forall c \in \text{ClassOf}(destVar).\neg\text{HierarchyDefinesInstVar}(c, varName)$ must be checked. If it fails, the entire refactoring is undone. Otherwise, the instance variable is defined on the new class.

5.7 Feasibility of Dynamic Refactoring

It is generally agreed that every software development should have a test suite to ensure correctness [Bro80]. Moreover, recent discussions about integrating refactoring into a commercial development process stress the importance of having a test suite before attempting any refactoring, whether manual or automatic [GHH97][Fow99]. Therefore, we do not think that it is unreasonable to assume the presence of a test suite. Projects without test suites probably have problems that will not be solved by refactoring.

Dynamic refactoring is no worse than manual refactoring and is often better. When refactoring manually, the programmer must update all of the references by hand. If the programmer misses any, the test suite will fail. The programmer will correct the error, and run the test suite again. With dynamic refactoring, the test suite must still be executed to perform the analysis, but it will complete successfully and will not have to be run again for this refactoring. If, later, a customer finds an error in the program that was introduced by the refactoring, it is evident that the refactoring was incorrect or incomplete and that the test suite was inadequate. In both the manual and dynamic case, the test suite should be augmented to catch the error. With the dynamic approach, the quality of the analysis can be measured by test quality metrics such as coverage.

To show that this approach is no worse than the manual approach, consider the following scenario. A programmer refactors the program using a dynamic refactoring and runs the test suite. He then releases the program. After the release, a user finds an error. Was the error introduced by the refactoring, or was it present before? The test suite would not have caught it if it was present, so it is not possible to determine if it was preexisting. Either way, the test suite should be extended to catch the particular error. Manual refactoring will have the same possibility of error along with the added tedium of having to perform the refactoring by hand.

The dynamic approach to refactoring has not yet been incorporated into the publicly available tool. We feel that the testing tool must be better integrated with the refactoring tool to make this approach acceptable to commercial programmers. We intend to make such an integration in future releases of the tool and determine if such an approach will be adopted by programmers.

Chapter 6

Refactoring Tools

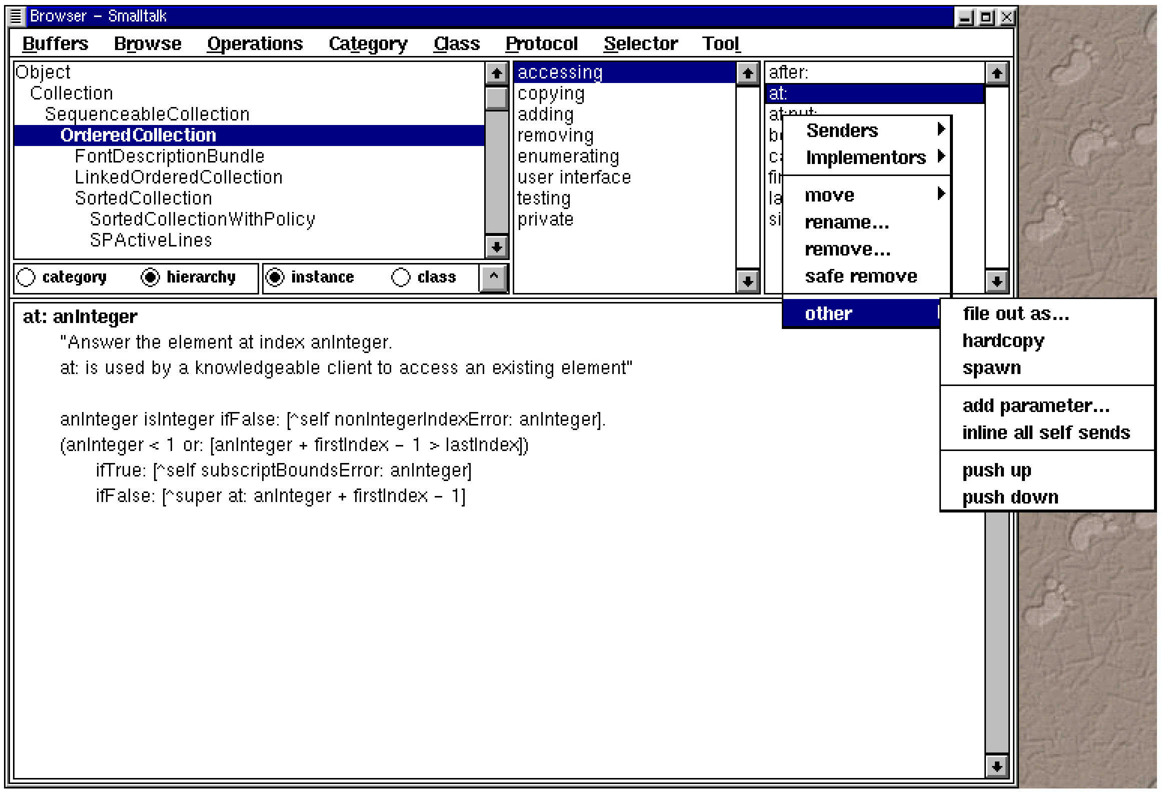
The goal of this research has been to bring program transformation and analysis into the realm of real-world languages and programs. As part of this research, we developed the Refactoring Browser, a tool to refactor Smalltalk programs [JBR95, RBJ97]. Developing this tool has given us insight into the criteria for a successful tool and given us a framework with which to experiment on more radical forms of program transformation tools.

6.1 The Refactoring Browser

There were at least two reasons for developing the Refactoring Browser. First, we wanted to create a *practical* refactoring tool that could be used in commercial software development. We felt that many of the ideas of evolutionary software development, such as continual refactoring and designing to the current set of requirements, would not be fully realized until we had produced such a tool. Secondly, we wanted a framework with which to test many of the newer refactoring ideas that we had. We feel that we have been successful on both fronts.

6.1.1 History

The first incarnation of the Refactoring Browser, then known simply as The Refactory, was a stand-alone tool separate from any of the other tools in the Smalltalk environment. It



```
at: anInteger
    "Answer the element at index anInteger.
    at: is used by a knowledgeable client to access an existing element"

    anInteger isInteger ifFalse: [^self nonIntegerIndexError: anInteger].
    (anInteger < 1 or: [anInteger + firstIndex - 1 > lastIndex])
        ifTrue: [^self subscriptBoundsError: anInteger]
        ifFalse: [^super at: anInteger + firstIndex - 1]
```

Figure 6.1: The Refactoring Browser

implemented many of the refactorings that exist in the current tool, but in a cruder fashion. While technically interesting, it was rarely used, even by ourselves. This led us to examine what the real criteria were to get programmers to use this tool. As a result, we created the Refactoring Browser.

Currently, the Refactoring Browser is being used on several commercial programming projects around the world in areas such as public health, insurance, telecommunications, and banking. The tool has performed reliably in many different contexts. Programmers have begun to realize the power that even simple automated refactorings give them. Nearly all of our feedback has been positive and most of the requests have been for additional features. Figure 6.1 shows a screenshot of the publicly-available tool.

6.1.2 The Research Framework

The Refactoring Browser has several components that are useful for research in program transformation. It has a custom-built parser for the Smalltalk language. This parser can accept an extended syntax to represent pattern nodes in parse trees. These trees can then be used as patterns for the parse tree rewriter component of the Refactoring Browser. The rewriter is the engine that actually performs the source-to-source transformations that implement the refactorings. There is also a framework for program analysis. The various conditions that are checked to verify the preconditions for the refactorings are reified as objects. These objects perform the necessary analysis to determine if the refactoring is valid. These objects facilitate the implementation of both the assertions and the dynamic analysis techinqies that this dissertation describes.

6.2 Success Criteria

Early versions of the refactoring tool were hardly used. The basic functionality for performing refactorings was present, but was insufficient for ensuring the success of the tool. As we developed later versions of the tool, we were forced to examine the issues that determined whether the tool was being used or not. This section examines the criteria that we identified as necessary for the success of a refactoring tool.

6.2.1 Technical Criteria for a Refactoring Tool

The main purpose of a refactoring tool is to allow programmers to easily refactor code. The first thing that programmers notice about automatic refactorings is that they reduce the amount of editing necessary to change the code. The programmer specifies the change that they want to make, and the refactoring changes the code quickly. When refactoring manually, the programmer must run the test suite after each small change to ensure that the program behaves the same as before the refactoring. Testing is time-consuming even when automated. Correct, automatic refactorings can reduce the need for that step, and can accelerate the refactoring process significantly.

The technical requirements for a refactoring tool are those properties that allow it to transform a program while preserving its behavior. This ensures that the test suite will not have to be rerun after every little change. Even dynamic refactoring, which requires running the test suite, reduces testing. With manual refactoring, the programmer makes a small change in the code, and runs the test suite. If the test fails, he corrects the error and reruns the test suite. Dynamic refactoring will only run the test suite once. Therefore time is saved both by eliminating the updating of cross-references manually, and only running the test suite once.

- **Program Database** One of the first requirements that we recognized was the ability to search for various program entities across the entire program. For example, a program might want to find all calls that can potentially refer to a particular method. Tightly integrated environments such as Smalltalk constantly maintain this database. At any time, the programmer can perform a search to find cross references. The maintenance of this database is facilitated by the dynamic compilation of the code. As soon as a change is made to any class, it is immediately compiled into bytecodes and the database is updated. In more static environments such as Java, the code is entered into text files. Updates to the database must be performed explicitly. These updates are very similar to the compilation of the Java code itself. Some of the more modern environments such as IBM's VisualAge for Java mimic Smalltalk's dynamic update of the program database.

  There is a range of techniques that can be used to construct this information. At one end of the scale are fast, lexical tools such as grep. At the other end are sophisticated analysis techniques such as dependency graphs. Somewhere in the middle is syntactic analysis using abstract syntax trees. There are tradeoffs between speed, accuracy, and richness of information that must be made when deciding which technique to use. For instance, grep is extremely fast but can be fooled by things like commented-out code. Dependency information is useful, but often takes a considerable amount of time to compute.

- **Abstract Syntax Trees (ASTs)** Most refactorings have to manipulate portions of the system that are below the method level. These are usually references to program elements that are being changed. For example, if an instance variable is renamed (simply a definition change), all references within the methods of that class and its subclasses must be updated. Other refactorings are entirely below the method level, such as extracting a portion of a method into its own, stand-alone method, or assigning a common subexpression to a temporary variable and replacing all occurrences of the expression with the temporary. Any update to a method needs to be able to manipulate the structure of the method. To do this requires ASTs.

  There are more sophisticated program representations that contain more information about the program such as control flow graphs [ASU88] and program dependency graphs [FOW87, KKL$^+$81]. However, we have found the ASTs contain sufficient information to implement powerful refactorings and are created extremely quickly. Morgenthaler found that the information that was contained in CFGs and PDGs could be extracted from ASTs as it was needed rather than having to create the additional graphs [Mor97]. If we require more sophisticated information in the future, we may adopt some of his techniques.

- **Accuracy** The refactorings that a tool implements must reasonably preserve the behavior of programs. Total behavior-preservation is impossible to achieve, as argued in Chapter 3. For example, a refactoring might make a program a few milliseconds faster or slower. Usually, this would not affect a program, but if the program requirements include hard real-time constraints, this could cause a program to be incorrect. Even more traditional programs can be broken. For example, if a program constructs a String and then uses the language's reflective facilities to execute the method that the String names, renaming the method will cause the program to have errors.

However, refactorings can be made reasonably accurate for most programs. Most programs do not rely extensively on techniques that defy our analysis and most do not have hard timing constraints. In the documentation of our tool, we have identified the cases that will break a refactoring so programmers that use those techniques can either avoid the refactoring or manually fix the parts of the program that the refactoring tool cannot fix. One of the techniques that we have used to increase the accuracy of the tool is to let the programmer provide information that would be difficult to compute. For example, in the publicly available version of the tool, when the type of a variable is required, we perform a quick-and-dirty type analysis based simply on the set of messages sent to the variable. This list often contains extraneous classes, and sometimes does not include the correct class. We allow the user to select the correct classes from the list and to add classes if necessary. This technique introduces the possibility that the programmer is wrong, and will therefore introduce an error into the program. This approach is still safer than manual refactoring because the tool handles the nonlocal edits. Additionally, the tool provides an undo facility, which is discussed in the next section, that allows the programmer to correct flawed refactorings quickly. Another benefit of this approach is that programmers feel that they have more control over the process and trust the tool more.

6.2.2 Practical Criteria for a Refactoring Tool

Tools are created to support a human in a particular task. If a tool does not fit the way people work, they will not use it. The most important criteria are the ones that integrate the refactoring process with other tools.

- **Speed**

  The analysis and transformations that are required to perform refactorings can be time consuming if they are very sophisticated. The relative costs of time and accuracy must always be considered. If a refactoring takes too long, a programmer will never use the automatic refactoring, but will just perform it by hand. “Too long” depends upon the types of delays a programmer is used to. For instance, in Smalltalk, compilation of a method takes less than a second when a method is accepted. Therefore, programmers are not willing to wait for extended periods of time for a refactoring since they could perform it and test it in a matter of minutes. C++ programmers, on the other hand, are used to compiles taking several minutes or more. We believe that they would be more willing to accept refactoring tools that were slower, as long as it was faster to refactor the program with the tool than by hand.

  When you implement a refactoring, you should always consider speed. The Refactoring Browser has several refactorings, such as *Move Instance Variable to Component* and *Common Subexpression Elimination*, that we have not implemented simply because we don’t know how to implement them safely in a reasonable amount of time. Adding the postconditions discussed in this dissertation to the publicly available tool will allow us to add some of these more expensive refactoring to it. Another approach to consider if an analysis would be too time consuming is to simply ask the programmer to provide the information. This puts the responsibility for accuracy back into the hands of the programmer while still allowing the refactoring to be performed quickly. The next criterion deals with handling incorrect information.

- **Undo** As mentioned earlier, automatic refactoring allows an exploratory approach to design. You can push the code around and see how it looks under the new design. Since a refactoring is supposed to be behavior-preserving, the inverse refactoring, which undoes the original, is also a refactoring and behavior-preserving. Earlier version of the Refactoring Browser did not incorporate the undo feature. This made refactoring a little more tentative because undoing some refactorings, while behavior-preserving, was difficult. Quite often I would have to go find an old version of the program and start again. This was annoying. With the addition of undo, yet another fetter was thrown off. Now I can explore with impunity, knowing that I can roll back to any prior version. I can create classes, move methods into them to see how the code will look, change my mind and go in a completely different direction, all very quickly. Other researchers have observed the necessity of an undo operation in tools that are used in an exploratory manner [ACS84, BG97].

- **Integrated with Environment** In the past decade the Integrated Development Environment has been at the core of most development projects. The IDE integrates the editor, compiler, linker, debugger, and any other tools necessary for developing programs. An early implementation of the refactoring tool for Smalltalk was a stand-alone tool. What we found was that no one used it, in fact, we did not even use it ourselves. Once we integrated the refactorings directly into the Smalltalk Browser, we used them extensively. Simply having them at your fingertips made all the difference.

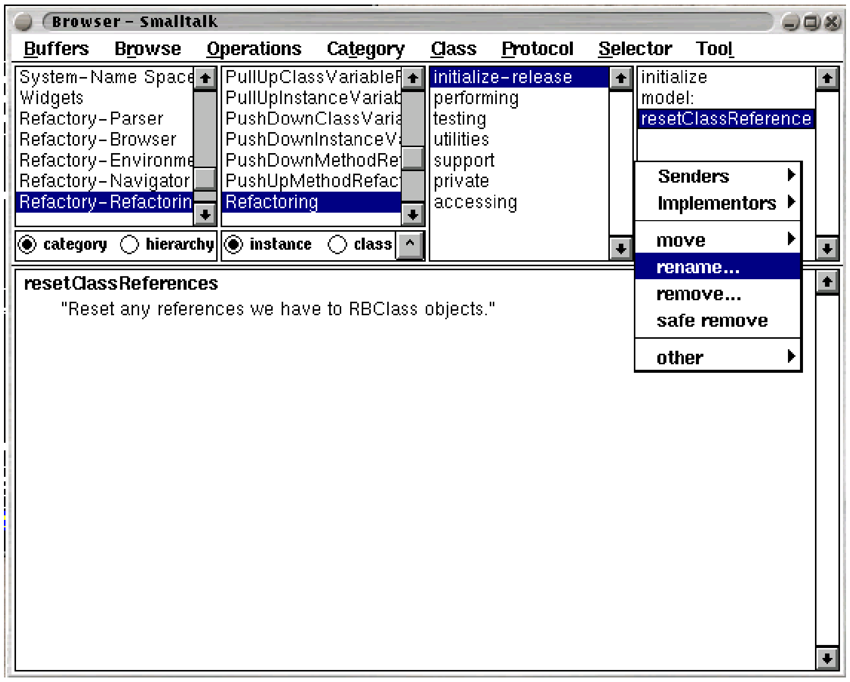
Chapter 7

The Architecture of the Refactoring Browser

As we developed the Refactoring Browser, we discovered a reusable framework for browsing and transforming code. We did not explicitly design this architecture from the beginning, but rather allowed it to evolve. This chapter describes the components of that design. John Brant probably wrote more lines of code than I did, and is the main author of some of these components, but I will describe it as a unified whole, rather than focusing on who did what.

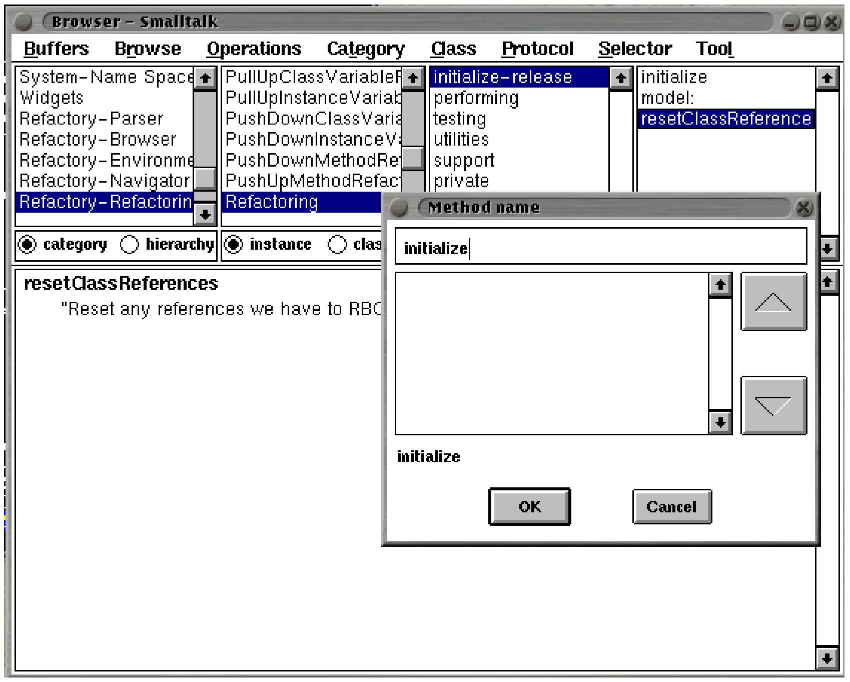
7.1 Example Refactoring

To demonstrate how the portions of the Refactoring Browser fit together, we will use the *rename instance variable* refactoring. To rename an instance variable using the Refactoring Browser, select the menu item from the Class menu, "Rename Instance Variable...". From the resulting dialog, select the instance variable to rename and then type in the new name. The browser will check that the preconditions are valid. If they are valid, it will perform the refactoring. Otherwise, it complains about the illegal condition and aborts the refactoring. Figures 7.1, 7.2, and 7.3 shows the series of screenshots that occurs whenever an illegal rename occurs.



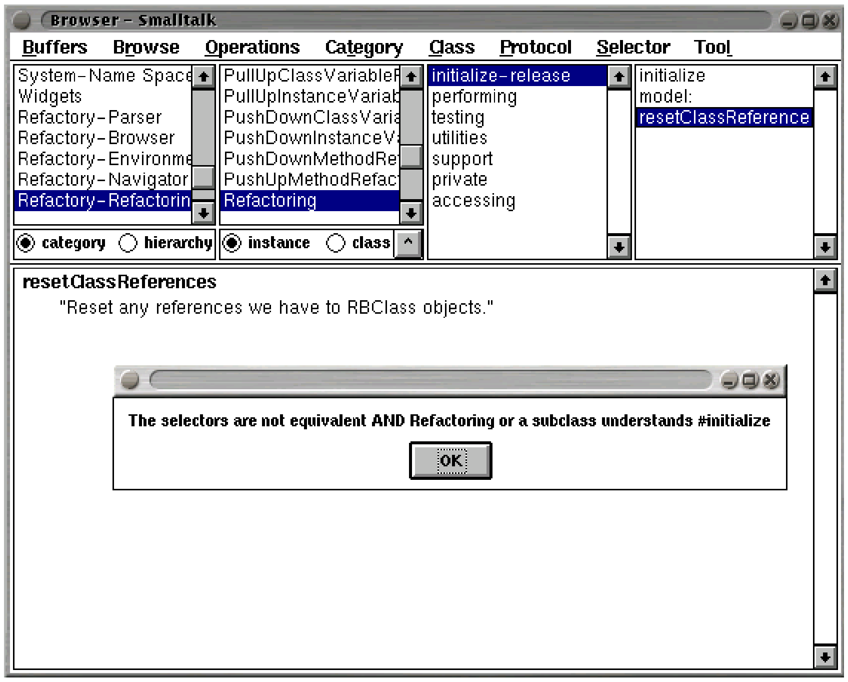
```
resetClassReferences
    "Reset any references we have to RBClass objects."
```

Figure 7.1: Menu Selection for Rename Method Refactoring



```
resetClassReferences
    "Reset any references we have to RBC
```

Figure 7.2: Enter the New Selector for a Rename Method Refactoring



```
resetClassReferences
	"Reset any references we have to RBClass objects."
```

The selectors are not equivalent AND Refactoring or a subclass understands #initialize

Figure 7.3: Illegal Rename Method Refactoring

7.2 The Transformation Framework

There are six parts to the transformation framework: the refactorings, the conditions, the parser, the tree rewriter, the formatter, and the change objects.

7.2.1 Refactorings

The publicly-available version of the Refactoring Browser implements all of the refactorings shown in Table 7.1. There are several refactorings that we have not discussed because they are either local in scope (e.g., extracting a method), or are similar to another refactoring (e.g., adding a parameter is similar to renaming a method).

Each refactoring is implemented by a subclass of Refactoring. Each subclass must implement two methods; preconditions, which returns the Condition object that represents the preconditions that must be met for the refactoring to be legal, and performRefactoring, which carries out the refactoring by using RefactoryChange objects. To transform the structures

Add Class	Add Instance Variable
Remove Class	Remove Instance Variable
Rename Class	Rename Instance Variable
Remove Method	Abstract Instance Variable
Rename Method	Create Accessors for Instance Variable
Add Parameter to Method	Add Class Variable
Remove Parameter from Method	Remove Class Variable
Rename Temporary	Rename Class Variable
Inline Temporary	Abstract Class Variable
Convert Temporary to Instance Variable	Create Accessors for Class Variable
Extract Code as Temporary	Convert Superclass to Sibling
Extract Code as Method	Inline Call
Push Up/Down Method	Push Up/Down Instance Variable
Push Up/Down Class Variable	Move Method to Component
Convert Instance Variable to ValueHolder	Protect Instance Variable
Move Temporary to Inner Scope	

Table 7.1: Refactorings Implemented in the Refactoring Browser

within methods, we use our own Smalltalk parser and tree rewriter. The *rename instance variable* refactoring is implemented by the class RenameInstanceVariableRefactoring. The following sections discuss in detail the various components that are used to implement the two methods that every subclass of Refactoring must implement.

7.2.2 Conditions

Early versions of the refactorings had their preconditions hard-coded into methods. As we developed more refactorings, we began to see common tests that were required by several of them. Additionally, we began to see the value of asserting conditions to eliminate program analysis. As a result, we created Condition objects to represent the various analyses that the refactorings need to perform. These correspond to the analysis functions described in this dissertation. Table 7.2 lists the creation methods to create the various condition objects. In addition to these standard conditions, you can create an arbitrary condition using the withBlock: or withBlock:errorString: creation methods. These take a block that must evaluate

canUnderstand: aSelector in: aClass
definesClassVariable: aString in: aClass
definesInstanceVariable: aString in: aClass
definesSelector: aSelector in: aClass
directlyDefinesClassVariable: aString in: aClass
directlyDefinesInstanceVariable: aString in: aClass
hasSubclasses: aClass
hasSuperclass: aClass
hierarchyOf: aClass canUnderstand: aSelector
hierarchyOf: aClass definesVariable: aString
hierarchyOf: aClass referencesInstVar: aString
isAbstractClass: aClass
isClass: anObject
isEmptyClass: anObject
isGlobal: aString in: aRBSmalltalk
isImmediateSubclass: subclass of: superClass
isMetaclass: anObject
isSymbol: aString
isValidClassName: aString
isValidClassVarName: aString for: aClass
isValidInstanceVariableName: aString for: aClass
isValidMethodName: aString for: aClass
referencesInstVar: aString in: aClass
subclassesOf: aClass referToSelector: aSelector

Table 7.2: Instance Creation Methods for Condition

to true or false.

There is only one class that represents all of these conditions. The object contains the block to be evaluated, the error string that is printed if the condition fails, and the type of the Condition. When the condition is sent the check method, it evaluates the block and the returns the result. The type of a condition is the name of the function that it is checking along with all of the arguments that are used to create it. The type allows use to test equality between any two Condition objects.

Reifying the conditions in this manner allows for two important enhancements to the Refactoring Browser. First, it allows for asserting postconditions. By adding the asserted condition objects to a database, later refactorings can see if their preconditions have been pre-

```
RenameInstanceVariableRefactoring>>preconditions
    ↑(Condition isValidInstanceVariableName: newName for: class)
        & (Condition definesInstanceVariable: varName in: class)
        & (Condition hierarchyOf: class definesVariable: newName) not
        & (Condition isGlobal: newName) not
```

Figure 7.4: Precondition Code for the RenameInstanceVariable Refactoring

viously asserted, and avoid performing the analysis directly on the program. This database must maintain consistency by removing the negation of a condition whenever it is asserted, or vice versa. For instance, if an earlier refactoring in a sequence asserted $\neg$IsClass(Foo) and a later refactoring adds the assertion IsClass(Foo), the earlier assertion must be removed to maintain consistency. Not only can assertions be created this way, but hard-to-compute analysis functions, such as ClassOf can either be provided by earlier refactorings, or if computed, can be cached, resulting in a speedup. Second, it provides components that, given a good user-interface, could support user-defined refactorings. Users could compose new refactorings by combining the appropriate condition objects together with a program transformation specified in the extended Smalltalk syntax.

In addition to simple condition objects, we created conditions to represent the logical combination of conditions. By creating *and*, and *not* conditions, we are able to construct the complete precondition for a refactoring and evaluate it all at once.

The preconditions of the *rename instance variable* refactoring are fairly simple. Basically, the new variable name must be valid and not be defined anywhere in the hierarchy. Figure 7.4 shows the code that performs these checks.

7.2.3 The Parser

Our parser is a standard parser for Smalltalk augmented with the ability to parse a syntax that includes pattern variables. Original versions of the Refactoring Browser were only developed in VisualWorks Smalltalk, so we simply co-opted some extended syntax from the

Smalltalk parser that wasn't being used, and added the necessary behavior to the existing nodes to represent pattern variables. The syntax was cumbersome, but it worked.

As we targeted other dialects of Smalltalk, some of which did not allow access to the parser, it became apparent that we needed to write our own parser. This is more important than it seems. One of the fundamental difficulties of source-to-source transformations that Opdyke identified was the preservation of non-semantic information about the program text (e.g., comments, whitespace, formatting, etc.) [Opd92]. Traditional parsing techniques do not deal with maintaining this information since parsers are usually the front-end of compilers. Since this information has no effect on the code, it is ignored. Our parser attaches information to the nodes that it creates that identifies where comments were located. Currently, we do not store information about the format of the code, leaving that to the Formatter class.

Another inspiration for creating a new parser and scanner for Smalltalk was the poor design of the original. In the original, Parser was a subclass of scanner. This was evidently so they could pass information between them as instance variables. This information was also stored in a non-object-oriented fashion. For example, token classes were represented by symbols stored in one instance variable along with the value of the token stored in another. We believe that the only reason that this design persisted for as long as it did was that the syntax of the language has changed very little in the past several years.

Our criteria for the parsing component were the following:

- It had to be portable. Since we were wanting to port the Refactoring Browser to other dialects of Smalltalk, we wanted to be able to use a common foundation on all of them.

- It had to be nearly as fast as the existing parser. Transforming large systems (like the Smalltalk image) can potentially require parsing the source for the entire system. This had to be as fast as possible.

Based on these requirements, we decided to forgo parser or scanner generators and manually create a recursive-decent parser. This was relatively easy due to the simple syntax of Smalltalk. Since we were rewriting an ugly part of the system, we wanted to improve it. So, in addition to our requirements, we wanted to meet the following design goals:

- It must have classes not only for the AST nodes, but also for the individual tokens produced by the scanner. This allows us to attach additional information to the token itself regarding the actual source that produced the token. This makes the architecture much simpler than if more primitive types such as Symbols or Strings were used. With primitive types, additional information must be stored somewhere apart from the representation of the token, making for a complicated interface between the scanner and parser.

- It needed to exhibit good object-oriented design principles. Conceptually, a scanner is simply a stream of token objects. Therefore, a scanner should adhere to the standard Stream protocol. A parser should be an example of the Builder pattern[GHJV95] that takes the stream of tokens and builds an abstract syntax tree.

We were successful in all of our goals. The resulting parser was only a few percent slower than the stock VisualWorks parser, and 30% faster than the stock VisualAge Smalltalk parser. The major speedup was that pattern matching was now twice as fast as before due to the specialized pattern nodes that did not exist in the original parser.

Probably the biggest advantage of creating our own parser was that we were able to add syntax to the language to represent entire subtrees by pattern variables. By parsing this extended Smalltalk code, we can create pattern trees that are used by a tree rewriter to transform code. Since the syntax for these patterns is primarily Smalltalk, we are able to expose this capability to the user to allow them to perform their own advanced searches and replaces. The next section will discuss the matching algorithm and give some examples of matches.

For the *rename instance variable* refactoring, the parser must parse all of the methods that reference the instance variable being renamed. These methods are obtained by using the Environments discusses in section 7.3.2. The trees that the parser produces are passed to the parse tree rewriter for transformation.

7.2.4 Tree Rewriter

The tree rewriter is the part of the transformation framework that changes the individual methods. This section will look at the specification of patterns and the implementation of the rewriter.

Specification of Patterns

The parse tree rewriter takes a template in the form of a Smalltalk AST, which may contain named pattern variables, and a standard Smalltalk AST, and attempts to find a subtree that matches it. Alternatively, the rewriter can take a pair of pattern trees, which share a common set of pattern variables, and a standard Smalltalk AST, and replace every match of the first tree with the second with all of the pattern variables replaced by the corresponding subtree from the matched tree.

Every pattern variable must start with the backquote (‘) character to identify it. The simplest pattern variable (e.g., `‘foo`) will match any single variable. A pattern that would match any code that incremented a variable is `‘foo := ‘foo + 1`. Since both pattern variables in this pattern have the same name, they must be same variable in any code that matches this pattern. So the Smalltalk code, z := z + 1 would match the pattern, while x := y + 1 would not.

Usually, we want a more general match than a single variable. The `@` prefix allows a match on multiple items in a position. For example, the pattern `‘@foo` will match any expression no matter how complex. As an example, the pattern `‘@receiver add: ‘@item` will match all senders of the `add:` message.

With the syntax already discussed, only portions of statements can be matched. Often,

`	Pattern variable prefix
@	Match multiple items in this position
.	Match an entire statement
#	Match a literal
`	Recurse into subtree on a successful match

Table 7.3: Pattern Variable Prefixes

we need to match a statement or statements. To specify a pattern variable that matches a statement, use the period (.) prefix. If used in conjunction with the `@` prefix, the variable will match multiple statements. So, `` `.foo `` will match any single statement, while `` `.@foo `` will match any number of statements. The pattern

```
`@test ifTrue: [ `.firstStatement.
                 `.@trueStatements.]
       ifFalse: [ `.firstStatement.
                  `.@falseStatements.].
```

will match any conditional statement that has the same first statement in both branches.

It possible to restrict a pattern variable to only match a literal by using the `#` prefix. In Smalltalk, a literal is a number, string, character, symbol, or literal array. The pattern `` `@collection at: `#literal `` will match any code that is accessing a hard-coded location within a collection.

By default, the tree matching algorithm returns the topmost node that matches its pattern without looking at any of the subtrees of the match. So, for example, the pattern `` `@rec at: `@index `` when matched against the Smalltalk code array at: (otherArray at: 3)) would return one match corresponding to the leftmost at:. Obviously, in certain transformations, we will want to recurse into the subtrees of a match. To specify this, an addition backquote (`) is prefixed to the variable. If we wanted to find *all* occurrences of the at: message, the correct pattern would be ``` ``@rec at: ``@index ```. Table 7.3 gives the prefix characters that identify various types of pattern variables.

```
name → self name
name := ‘@obj → self name: ‘@obj
```

Figure 7.5: Transformation Rules for Abstract Instance Variable name.

```
BRAssignmentNode>>match: aNode inContext: aDictionary
    aNode class == self class ifFalse: [↑false].
    ↑(variable match: aNode variable inContext: aDictionary)
        and: [value match: aNode value inContext: aDictionary]
```

Figure 7.6: Matching Code for Assignment Nodes

To transform code, a target tree has to be specified using the same syntax. All of the pattern variables in the target tree must be present in the source tree. Figure 7.5 shows the transformation rules for abstracting all references to the variable name by replacing direct access with calls to getter and setter functions.

Implementation of the Rewriter

The rewriter is an instance of the Visitor pattern [GHJV95]. The Visitor pattern takes an algorithm that would normally be distributed across the various types of nodes in a data structure, and centralizes it within the visitor class. Each node then implements the accept: method that accepts the visitor as a parameter and calls a method on the visitor that corresponds to its type with itself as an argument. This is known as *double dispatching.* The rewriter first separates the nodes into argument nodes and other nodes. It then sends the match:inContext: to the node. The context that is passed to this method is the current assignment to the pattern variables. Figure 7.6 shows the matching code for an assignment node. Figure 7.7 shows the matching code for a pattern variable.

The transformation that must be performed to implement the *rename instance variable* refactoring is quite simple. The matching pattern is just the name of the variable to be rename and the rewrite pattern is just the new name. No pattern variables are necessary for this particular refactoring.

```
BRMetaVariableNode>>match: aNode inContext: aDictionary
    self isAnything ifTrue: [↑(aDictionary at: self ifAbsentPut: [aNode]) = aNode].
    self isLiteral ifTrue: [↑self matchLiteral: aNode inContext: aDictionary].
    self isStatement
        ifTrue: [↑self matchStatement: aNode inContext: aDictionary].
    aNode class == self matchingClass ifFalse: [↑false].
    ↑(aDictionary at: self ifAbsentPut: [aNode]) = aNode
```

Figure 7.7: Matching Code for Pattern Variable Nodes

In addition to implementing refactorings, we have used the searching capability of the tree matcher to create a tool to detect various types of common Smalltalk coding and style errors. By creating sets of patterns that correspond to the way that programmers commonly write the wrong thing, we are able to detect many errors that have never been found by test cases or users.

7.2.5 Formatter

Since the methods of the program are transformed by using the rewriter to perform the tree-to-tree transformation, we have to have some way to get the source back from the tree. To do this, we have implemented printString on ASTs to return nicely formatted code. This code is then passed to an AddMethodChange object, which is discussed in the next section.

To create the nicely formatted code from the parse tree, we use a class named BRFormatter. BRFormatter is a Visitor that traverses the AST and produces formatted code according to the style rules found in Kent Beck's book, "Smalltalk Best Practices Patterns" [Bec96]. Formatting code is a topic of much debate. As Kent notes in his book: "No discussion has generated more heat and less light." To help alleviate this, the way code is formatted can be customized by modifying or subclassing the BRFormatter class.

```
RenameInstanceVariableRefactoring>>performRefactoring
        | changeBuilder replacer |
        changeBuilder := CompositeRefactoryChange
                        named: 'Rename instance variable'.
        changeBuilder addInstanceVariable: newName to: class.
        replacer := ParseTreeRewriter
                        rename: varName
                        to: newName
                        handler:
                                [self
                                        refactoringError: ('<1s> is already defined as a method or
                                                block temporary <n> variable in this class or one of
                                                its subclasses'
                                                expandMacrosWith: newName)].
        self
                convertClasses: class withAllSubclasses
                select: [:aClass | aClass whichSelectorsAccess: varName]
                using: replacer
                notifying: changeBuilder
                message: 'Changing instance variable name from: ' , varName , ' to: '
                        , newName.
        changeBuilder removeInstanceVariable: varName from: class.
        self performChange: changeBuilder withLabel: 'Compiling changes'.
```

Figure 7.8: Code to Construct and Perform the Change for the RenameInstanceVariable Refactoring

7.2.6 Changes

All of the refactorings are implemented by using RefactoryChange objects. These objects are examples of the Command pattern and represent the low-level changes to the code [GHJV95]. Table 7.4 shows the twelve subclasses of RefactoryChange that are used to implement all the refactorings in the browser.

For the *rename instance variable* refactoring, we must add the new variable, transform all of the methods that reference the variable, and then remove the old variable. Figure 7.8 shows the code that creates the CompositeRefactoringChange that implements this refactoring.

Each RefactoryChange object must implement the asUndoOperation method that returns

CompositeRefactoringChange	Represents a composite change
RenameClassChange	Change the name of a class along with all references
AddClassChange	Add a class to the system
RemoveClassChange	Remove a class from the system
AddMethodChange	Add a method to the system
RemoveMethodChange	Remove a method from the system
AddClassVariableChange	Add a class variable to a class
AddInstanceVariableChange	Add an instance variable to a class
AddPoolVariableChange	Add a pool variable to a class
RemoveClassVariableChange	Remove a class variable from a class
RemoveInstanceVariableChange	Remove an instance variable from a class
RemovePoolVariableChange	Remove a pool variable from a class

Table 7.4: Subclasses of RefactoryChange

another RefactoryChange object that implements the inverse change on the program. Some of the changes in the table are obvious inverses such as the add and remove changes. The AddMethodChange allows new code to be compiled over old code. To implement undo for this change, there are two cases. If the method being added does not exist, the inverse operation is simply to remove the method. If the method already exists, the change must save the old code and the inverse operation is to recompile the old code. The CompositeRefactoringChange implements undo by keeping a list of the undo operations of its constituent changes in reverse order.

Since they are undoable both individually and as a composite, the change objects are extremely powerful. Every change that is made to the program by a refactoring is ultimately made by a sequence of change objects that can undo themselves. Several times we have implemented a convoluted refactoring that consisted of several different transformations, and been able to undo it immediately as soon as we were finished. Without change objects, implementing the refactoring would only be half the job, with undo being the other half.

For transformations that require changes below the method level, the system creates transformed code using the parser, tree rewriter and formatter. It then creates an AddMethodChange with the new code which is compiled over the original method. By ensuring that each of these changes can undo itself reliably, we can ensure that complex refactorings can be undone reliably. Additionally, by combining these changes with the condition objects, we can allow developers to define their own refactorings with a minimum of programming. We hope to provide an interface for this capability in future releases of the tool.

7.3 The Code-Browsing Framework

After the failures of the initial version of the refactoring tool, we decided that the best strategy for getting refactoring technology into the hands of real programmers was to develop a browser that eliminated many of the shortcomings of the existing browsers and include the refactorings. This "Trojan horse" approach unexpectedly led to the development of an entire framework for browsing code. This framework has been used as the front-end for such diverse software tools as profilers, style checkers, and testing frameworks. Figure 7.9 shows a graphical representation of the relationships between the classes in the code browsing framework.

7.3.1 RefactoringBrowser

The class RefactoringBrowser is a Façade[GHJV95] whose principal responsibility is to hold onto the other components of the browsing framework. It also provides the interface for creating a new Refactoring Browser by implementing the open method on the class side. RefactoringBrowser holds onto a Navigator that displays the current item being browsed. It also holds onto one or more CodeModels that display the item selected in the Navigator. One of the features that we added to the standard browser is the concept of **buffers**. Smalltalk programmers that use the standard system browsers find that they must open several windows to accomplish their tasks. To alleviate this problem, we allow multiple buffers to be created within the same window. These serve the same purpose as multiple windows, but with less clutter.

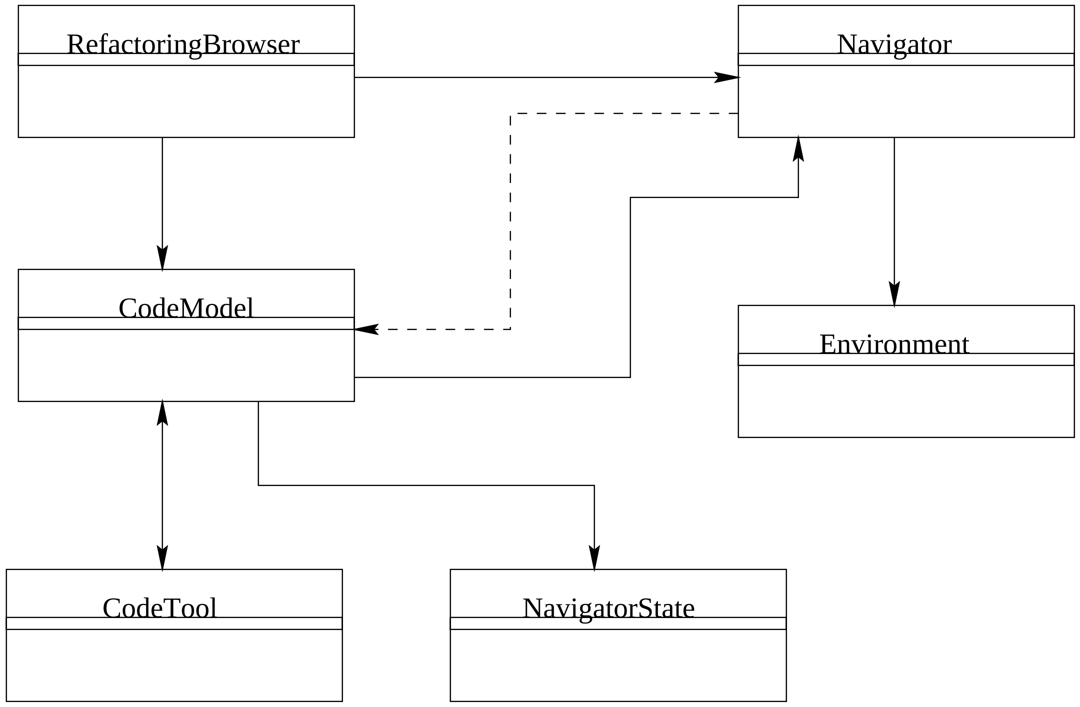


Figure 7.9: The Architecture of the Refactoring Browser

7.3.2 Environments

The fundamental data structure in the code-browsing framework is known as an *environment*. An environment is an arbitrary collection of classes and methods. An environment can be used to restrict the scope of the system which is searched during various queries such as finding all senders of a message. They have also been used as filters for displaying results from a style tool. They can also be used as a basis for metrics tools. They provide methods such as numberSelectors and numberClasses that return counts of the different entities within them. As of now, refactorings ignore environments and perform their transformations and analysis on the entire system. This has primarily been in the interest of safety. It is possible to use them to restrict the scope of transformations and with the recent introduction of namespaces in several dialects of Smalltalk, we will probably use them to restrict the scope of refactorings to single namespaces.

The common superclass of all environments is BrowserEnvironment. A new BrowserEnvi-

forCategories: categoryList	A set of categories
forClass: aClass protocols: protocolCollection	A set of protocols in a class
forClass: aClass selectors: selectorCollection	A set of selectors in a class
forClasses: classCollection	An arbitrary set of classes
implementorsMatching: aString	Implementors matching aString
implementorsOf: aSelector	Implementors of selector
instVarReadersTo: instVarName in: aClass	The methods that read an instance variable
instVarRefsTo: instVarName in: aClass	The methods that reference an instance variable
instVarWritersTo: instVarName in: aClass	The methods that write to an instance variable
matches: aString	Methods that contain a literal matching aString
referencesTo: aLiteral	Methods that reference a literal
referencesTo: aLiteral in: aClass	Methods that references a literal within a class
selectMethods: aBlock	Arbitrary selection of methods based on aBlock

Table 7.5: Environment Restriction Methods

ronment contains all of the classes and methods of the system. Other, more restrictive, kinds of environments are created by sending messages to an existing environment. Table 7.5 lists the methods that restrict an environment. They are always sent to an existing environment to restrict it according to some criteria. For instance, to create an environment restricted to a set of classes, you would use the statement: env := BrowserEnvironment new forClasses: classes.

All environments that are restricted somehow are subclasses of BrowserEnvironmentWrapper. A BrowserEnvironmentWrapper has an instance variable, environment, that points to the environment that it limits. This is an example of the Decorator pattern [GHJV95]. We have specialized subclasses of BrowserEnvironmentWrapper that represent some of the more common queries. These are CategoryEnvironment, which represents all of the classes and methods within a set of VisualWorks categories; ClassEnvironment, which represents all of the methods within an arbitrary set of classes; and ProtocolEnvironment, which represents all of the methods within a set of VisualWorks protocols. The class SelectorEnvironment represents an arbitrary collection of classes and selectors. These environments are not usually created by sending a creation method to the class itself, but are created by sending one of

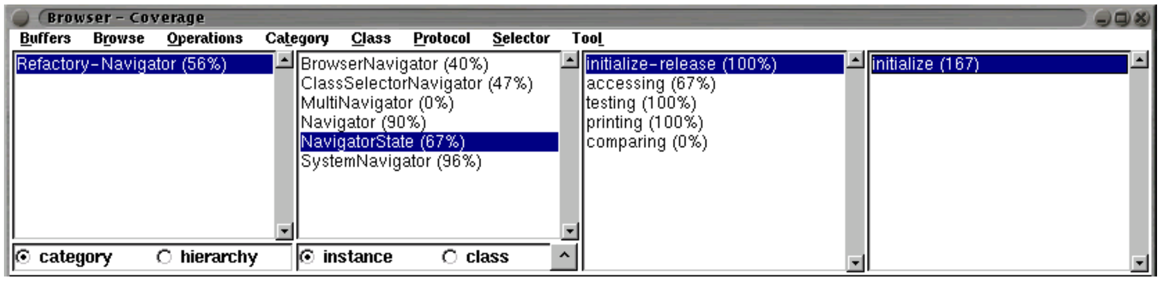


Figure 7.10: Navigator Customized as Front-End to a Coverage Tool

the messages in Table 7.5 to an existing environment. Since these can be used to restrict *any* environment, complex queries can be constructed.

Two additional environments, NotEnvironment and AndEnvironment provide for the logical combination of queries. All environments can be combined using the logical operators &, |, and *not*. These operations create a combination of AndEnvironments and NotEnvironments to represent the query. The semantics of the NotEnvironment is to return every class and method in the system that is not in the environment that it is placed on. We create the *or* operation by using DeMorgan's law that states $A \vee B = \neg(\neg A \wedge \neg B)$.

7.3.3 Navigator

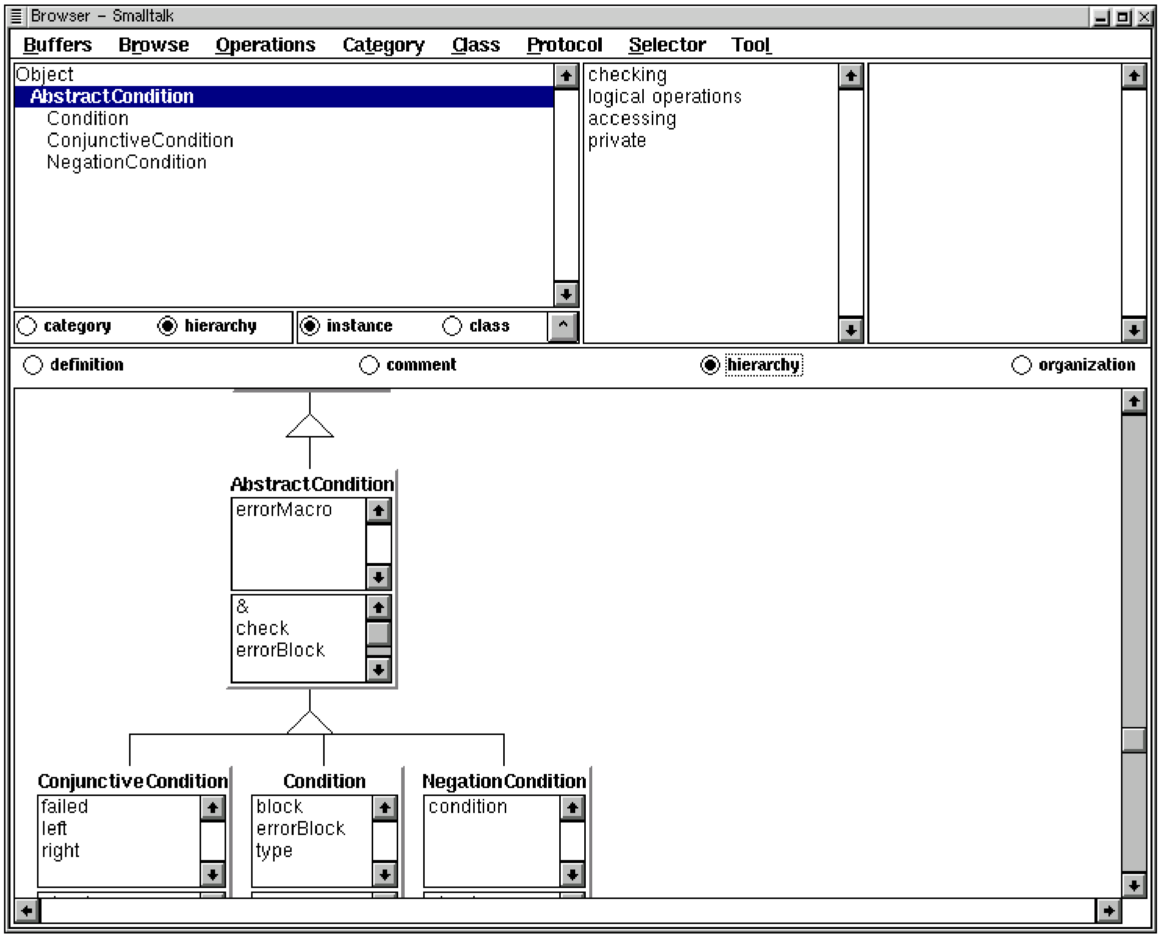
One of the fundamental activities that occurs while browsing code is selecting a class and methods. The user interface to an environment that facilitates browsing is known as a *navigator*. The navigator is subclassed to change the way the tool displays the elements of the environment or to add commands to operate on the elements of the environment. Figure 7.10 shows a screenshot of a navigator that has been customized to be the front end of a coverage tool. Commands take the form of menu items that appear in the GUI. There are some menu items such as *Find Class...* and *Find Method...* that are standard.

7.3.4 CodeModel and NavigatorState

The CodeModel class is fairly simple. It is responsible for determining which CodeTool to use when displaying the current selection within the Navigator. It does this by being a dependent of the Navigator. Whenever the selection in the Navigator changes, the CodeModel is notified and it takes the appropriate action. It also holds onto an instance of NavigatorState. NavigatorState is an example of the Memento pattern [GHJV95]. It contains all of the information necessary to specify the state of the current selection within the Navigator. Buffers are implemented by having multiple CodeModels, each containing its own NavigatorState. Whenever the user switches to a buffer, the corresponding CodeModel passes its NavigatorState object to the Navigator, which causes it to change the selection to match the one stored in the NavigatorState object.

7.3.5 CodeTools

The Smalltalk Browser has traditionally consisted of several small panes at the top that are used to find the class and possibly the method to be browsed, and a lower portion that actually displays the code. In the Refactoring Browser, this lower portion is replaced by a CodeTool. A CodeTool is an arbitrary view whose type is based on the selection within the navigator. It is an example of the Strategy pattern [GHJV95]. By allowing arbitrary views, we can provide more specialized views on the code than the text-based views that were provided by the original browser. For example, when viewing a method that defines an icon, rather than displaying the literal array that encodes the icon, we display the icon itself by using an IconViewer, a subclass of CodeTool. We can also allow the user to select different views for the same entity. For example, when a class is selected, the user can view the class definition, the class comment, or a graphical representation of the hierarchy. Figure 7.11 shows an example of a code tool that displays a graphical view of the selected class's hierarchy.



Object
AbstractCondition
Condition
ConjunctiveCondition
NegationCondition

checking
logical operations
accessing
private

(● hierarchy) (● instance) (● hierarchy)

AbstractCondition
errorMacro
&
check
errorBlock

ConjunctiveCondition
failed
left
right

Condition
block
errorBlock
type

NegationCondition
condition

Figure 7.11: CodeTool that Displays a Graphical Hierarchy

Chapter 8

Conclusions

8.1 Summary of Contributions

This research has explored ways to make refactoring object-oriented programs more practical. The principal artifact of this research is a commercial-grade tool, the Refactoring Browser, that is suitable both for commercial software development, and as a platform for experimenting with new ideas about program transformation. In particular, the following contributions have been made:

1. A formal definition of refactoring that is based on postconditions.
2. Specification of several common refactorings with the new definition.
3. Design of a refactoring tool that reduces program analysis by using the postcondition assertions.
4. A method of calculating the preconditions for composite refactorings.
5. Definition of dependency between refactorings based on commutativity.
6. Formula for calculating the conditions under which any two refactorings may commute.
7. A set of applications that use the dependency information calculated from a chain of refactorings.

8. A scheme for using dynamically obtained information to perform refactorings.

8.2 Future Work

One obvious direction to extend this research is into other languages, such as Java. Every language has its own unique features that have an impact on refactoring and introduce new refactorings. For instance, Java's interfaces present an new set of refactorings along with constraints on existing refactorings.

We have been using the Refactoring Browser for quite a while and teaching others how to use it. We can create new refactorings quite easily with the frameworks that are in it, but it is difficult for other people to do it themselves. Therefore, another area of research is to create a visual environment that exposes the various parts of the transformation framework to users. This would allow them create their own refactorings or transformations easily constructing them from parts such as existing refactorings, new conditions, and arbitrary transformations. Figure 8.1 shows a sketch of what the interface might look like. The refactorings that they create could be general refactorings that are applicable to any code, or they might be refactorings that are specific only to a particular application.

We have discussed refactorings role in framework design and evolution. As a framework evolves, there will be application code that must be updated to the new design. If the framework is sold to clients, that application code may not be available for refactoring along with the framework. Ideally, the framework vendor could release a set of refactorings that clients could apply to their applications that would migrate their code to the new version of the framework. An area of research that we have not yet examined is creating these tools that support this.

As stated earlier, refactorings are typically applied in sequences to perform composite refactorings. The Refactoring Browser has defined a few of these composite refactorings already. Users should be able to define their own composite refactorings, similar to a macro

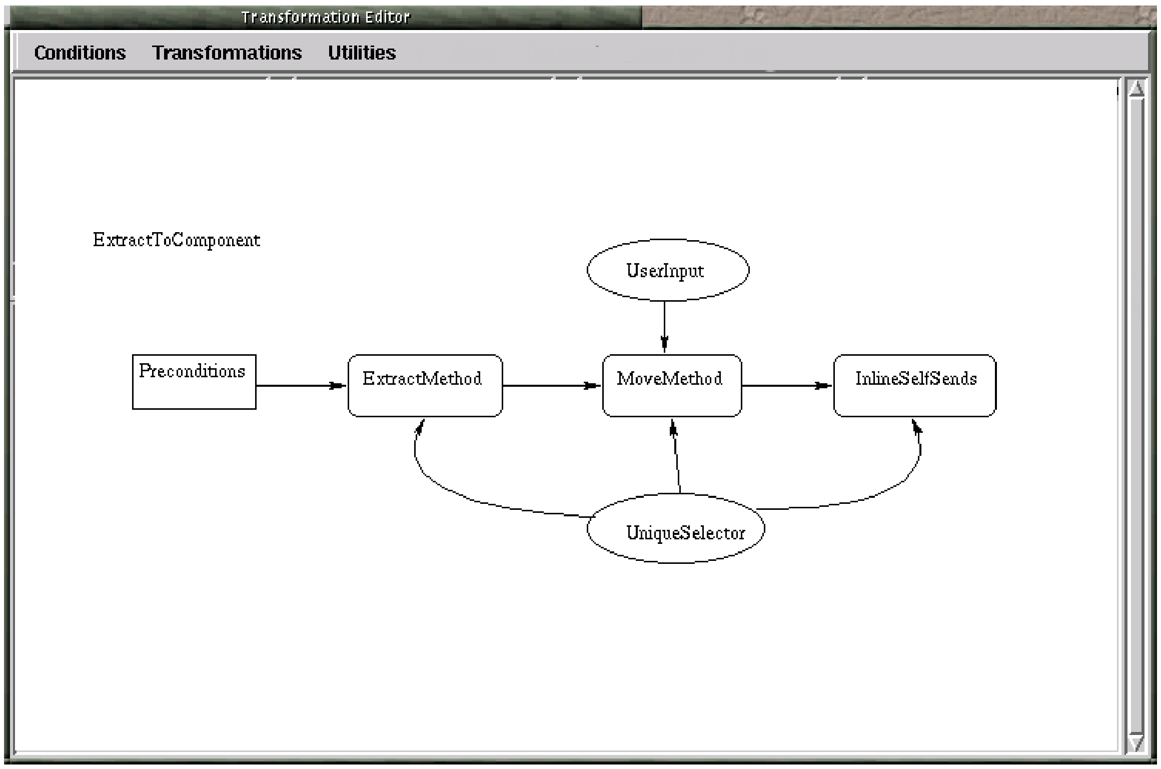


Figure 8.1: Hypothetical Tool to Allow User-Defined Refactorings

facility. The interesting parts of this would be in the interface to construct the composite. Which parameters of the refactorings within the composite are connected to the output of earlier refactorings, and which ones should be supplied by the user?

Since refactoring causes global changes to a program, refactoring in a multi-user environment presents a series of interesting problems. We have pointed out how to check conflicts between groups of refactorings, but that is only a part of the problem. Consider the case where programmer A checks out a portion of the system. Programmer B then performs a refactoring that affects the code the Programmer A has checked out. How is this resolved? Two possible solutions are that programmer A is notified via email of the changes that need to be performed in his code, or when A submits his changes, the refactoring is performed on them, possibly forcing an undo of the refactoring if the submitted code violates a precondition of the refactoring.

A more theoretical area of research would be to study some of the algebraic properties of refactorings. Since a refactoring is behavior-preserving, its inverse must also be a refactoring. However, some of the inverse refactorings are more difficult to perform because they require more difficult analysis. Suppose a class within your application has a method named `foo:` and no one else implements the `foo:` method. Renaming `foo:` to `add:` does not require any complex analysis since every sender of `foo:` must refer to the method being renamed. However, the inverse refactoring must discriminate among the many implementer of `add:` when attempting to change the name back to the original. We have scratched the surface with the commutativity property. Other interesting properties would be associativity, simplifications, and a minimal set of refactorings.

While all of these areas may be intellectually challenging, the ones that are most valuable must be useful to the programmers in the trenches. The goal of software engineering research is to make it easier to build correct software. We should examine the problems that face programmers and work with them to resolve those problems. Tools that support programmers in their day-to-day activities, such as the Refactoring Browser, allow them to realize a qualitatively different type of software development—one that recognizes that those involved will learn as they create the software and allow them to change earlier decisions. Clever analysis algorithms and expressive notations do not, in themselves, help the average programmer. Tools that help the average programmer are those that recognize the presence of an intelligent mind at the helm and attempt to support it.

Appendix A

Refactorings

In this appendix, only the analysis functions that are affected by the refactoring are shown. All of the other analysis functions are not affected by the refactoring. They have been elided for readability.

A.1 Class Refactorings

AddClass$(class, superclass, subclasses)$

$$\begin{array}{rl} pre: & \text{IsClass}(superclass) \\ & \wedge\neg\text{IsClass}(class) \\ & \wedge\neg\text{IsGlobal}(class) \\ & \forall c \in subclasses.(\text{IsClass}(c) \wedge \text{Superclass}(c) = superclass) \\ \hline post: & \text{IsClass}' = \text{IsClass}[class/true] \\ & \text{ClassReferences}' = \text{ClassReferences}[class/\phi] \\ & \text{Superclass}' = \text{Superclass}[class/superclass] \\ & \text{InstanceVariablesDefinedBy}' = \text{InstanceVariablesDefinedBy}[class/\phi] \end{array}$$

RemoveClass$(class)$

$$\begin{array}{rcl}
pre & : & \text{IsClass}(class) \\
 & & \wedge\text{ClassReferences}(class) = \phi \\
 & & \wedge(\text{Subclasses}(class) = \phi \vee \text{IsEmptyClass}(class)) \\
\hline
post & : & \text{IsClass}' = \text{IsClass}[class/false] \\
 & & \text{ClassReferences}' = \forall c \neq class.\forall s.\text{ClassReferences}[class/\perp] \\
 & & \qquad [c/\text{ClassReferences}(c) - \{(class, s)\}] \\
 & & \text{Superclass}' = \forall c \in \text{Subclasses}(class).\text{Superclass}[class/\perp][c/\text{Superclass}(class)] \\
 & & \text{Method}' = \forall s.\text{Method}[(class, s)/\perp] \\
 & & \text{Senders}' = \forall s.\text{Senders}[(class, s)/\perp] \\
 & & \text{ReferencesToInstanceVariable}' = \forall v.\forall c.\text{ReferencesToInstanceVariable}[(class, v)/\perp] \\
 & & \quad [(c, v)/\text{ReferencesToInstanceVariable}(c, v) - \{(s, t) | s = class\}] \\
 & & \text{InstanceVariablesDefinedBy}' = \text{InstanceVariablesDefinedBy}[class/\perp]
\end{array}$$

RenameClass$(class, newClass)$

$$
\begin{aligned}
pre \quad : \quad & \text{IsClass}(class) \\
& \wedge\neg\text{IsClass}(newClass) \\
& \wedge\neg\text{IsGlobal}(newClass) \\
\hline
post \quad : \quad & \text{IsClass}' = \text{IsClass}[class/false][newClass/true] \\
& \text{ClassReferences}' = \forall c \neq class . \forall s \in \{s|(class, s) \in \text{ClassReferences}(c)\}. \\
& \quad\quad \text{ClassReferences}[class/\perp][newClass/\text{ClassReferences}(class)] \\
& \quad\quad\quad [c/\text{ClassReferences}(c) - \{(class, s)\} \cup \{(newClass, s)\}] \\
& \text{ReferencesToInstanceVariable}' = \\
& \forall(c, v, s) \in \{(c, v, s)|(class, s) \in \text{ReferencesToInstanceVariable}(c, v) \wedge c \neq class\}. \\
& \forall lv . \forall lr \in \{lr|(class, lr) \in \text{ReferencesToInstanceVariable}(class, lv)\}. \\
& \quad\quad \text{ReferencesToInstanceVariable}[(class, lv)/\perp] \\
& \quad\quad\quad [(newClass, lv)/\text{ReferencesToInstanceVariable}(class, lv) \\
& \quad\quad\quad\quad -\{(class, lr)\} \cup \{(newclass, lr)\}] \\
& \quad\quad\quad [(c, v)/\text{ReferencesToInstanceVariable}(c, v) - \{(class, s)\} \cup \{(newClass, s)\}] \\
& \text{Superclass}' = \forall c \in \text{Subclasses}(class).\text{Superclass}[class/\perp] \\
& \quad\quad\quad [newClass/\text{Superclass}(class)][c/newClass] \\
& \text{Method}' = \forall c . \forall s . \text{Method}[(newClass, s)/\text{Method}(class, s)][(class, s)/\perp] \\
& \quad\quad\quad [(c, s)/\text{RenameReferences}(\text{Method}(c, s), class, newClass)] \\
& \text{Senders}' = \forall s . \text{Senders}[(newClass, s)/\text{Senders}(class, s)][(class, s)/\perp] \\
& \text{InstanceVariablesDefinedBy}' = \\
& \quad \text{InstanceVariablesDefinedBy}[newClass/\text{InstanceVariablesDefinedBy}(class)] \\
& \quad\quad [class/\perp]
\end{aligned}
$$

A.2 Method Refactorings

AddMethod$(class, selector, body)$

$$\frac{\begin{array}{rl} pre: & \text{IsClass}(class) \\ & \wedge\neg\text{DefinesSelector}(class, selector) \\ & \wedge(\neg\text{UnderstandsSelector}(class, selector) \\ & \quad\vee\text{LookedUpMethod}(class, selector) \overset{\alpha}{\equiv} body)\end{array}}{\begin{array}{rl} post: & \text{Method}' = \text{Method}[(class, selector)/body] \\ & \text{Senders}' = \text{Senders}[(class, selector)/\phi] \\ & \text{ReferencesToInstanceVariable}' = \forall c \in \text{Superclass}^*(class) \cup \{class\}. \\ & \quad \forall v \in \{v|\text{DefinesInstanceVariable}(c, v)\} \cap \text{UnboundVariables}(body). \\ & \quad \text{ReferencesToInstanceVariable}[(c, v)/ \\ & \quad \text{ReferencesToInstanceVariable}(c, v) \cup \{(class, selector)\}] \\ & \forall c \in \{c|\text{IsClass}(c)\} \cap \text{UnboundVariables}(body). \\ & \text{ClassReferences}' = \text{ClassReferences}[c/\text{ClassReferences}(c) \cup \{(class, selector)\}]\end{array}}$$

RemoveMethod$(class, selector)$

$$\begin{array}{rcl}
pre & : & \text{IsClass}(class) \\
& & \wedge\text{DefinesSelector}(class, selector) \\
& & \wedge(\text{Senders}(class, selector) = \phi \\
& & \quad \vee\text{Method}(class, s) \overset{\alpha}{\equiv} \text{LookedUpMethod}(\text{Superclass}(class), s)) \\
\hline
post & : & \text{Method}' = \text{Method}[(class, selector)/\perp] \\
& & \text{Senders}' = \text{Senders}[(class, selector)/\perp] \\
& & \forall c \in \{d|\text{IsClass}(d)\} \cap \text{UnboundVariables}(\text{Method}(class, selector)). \\
& & \text{ClassReferences}' = \text{ClassReferences}[c/\text{ClassReferences}(c) - \{(class, selector)\}] \\
& & \text{ReferencesToInstanceVariable}' = \forall c \in \text{Superclass}^*(class) \cup \{class\}. \\
& & \quad \forall v \in \{w|\text{DefinesInstanceVariable}(c, w)\} \cap \text{UnboundVariables}(body). \\
& & \quad \text{ReferencesToInstanceVariable}[(c, v)/ \\
& & \quad \text{ReferencesToInstanceVariable}(c, v) - \{(class, selector)\}]
\end{array}$$

RenameMethod$(classes, selector, newSelector)$

$$
\begin{array}{lcl}
pre & : & \forall c \in classes.\text{IsClass}(c) \\
& & \wedge \text{DefinesSelector}(c, selector) \\
& & \wedge \forall d \notin classes.\text{DefinesSelector}(d, selector) \\
& & \quad \rightarrow ((\text{Senders}(d, selector) \cap \text{Senders}(c, selector)) = \phi) \\
\hline
post & : & \text{Method}' = \forall d.\forall s.\forall c \in classes.\text{Method}[(c, selector)/\perp] \\
& & \qquad [(c, newSelector)/\text{Method}(c, selector)] \\
& & \qquad [(d, s)/\text{RenameReferences}(\text{Method}(d, s), selector, newSelector)] \\
& & \text{Senders}' = \forall c \in classes.\text{Senders}[(c, selector)/\perp][(c, newSelector)/\text{Senders}(c, selector)] \\
& & \text{ReferencesToInstanceVariable}' = \forall c \in classes.\forall (d, v) \in \\
& & \quad \{(d, v) | (c, selector) \in \text{ReferencesToInstanceVariable}(d, v)\}. \\
& & \quad \text{ReferencesToInstanceVariable}[(d, v)/(\text{ReferencesToInstanceVariable}(d, v) - \\
& & \quad \{(c, selector)\}) \cup \{(c, newSelector)\}] \\
& & \forall c \in \{d | \text{IsClass}(d)\} \cap \text{UnboundVariables}(\text{Method}(class, selector)). \\
& & \text{ClassReferences}' = \text{ClassReferences}[c/\text{ClassReferences}(c) \\
& & \qquad -\{(class, selector)\} \cup \{(class, newSelector)\}]
\end{array}
$$

MoveMethod$(class, selector, destVar, newSelector)$

$$
\begin{array}{rcl}
pre & : & \text{IsClass}(class) \\
& & \wedge \text{DefinesSelector}(class, selector) \\
& & \wedge \forall c \in \text{ClassOf}(class, destVar). \neg \text{UnderstandsSelector}(c, newSelector) \\
& & \forall c \in \text{Superclass}^*(class) \cup \{class\}. \forall v \in \text{InstanceVariablesDefinedBy}(c). \\
& & \quad (class, selector) \notin \text{ReferencesToInstanceVariable}(c, v) \\
\hline
post & : & \text{Method}' = \forall c \in \text{ClassOf}(class, destVar). \\
& & \quad \text{Method}[(class, selector)/\text{ForwarderMethod}(newSelector, destVar)] \\
& & \quad [(c, newSelector)/\text{MovedMethod}(\text{Method}(class, selector))] \\
& & \forall e \in \{d | \text{IsClass}(d)\} \cap \text{UnboundVariables}(\text{Method}(class, selector)). \\
& & \text{ClassReferences}' = \text{ClassReferences}[e/\text{ClassReferences}(e) - \{(class, selector)\} \\
& & \qquad \cup \{(c, newSelector)\}]
\end{array}
$$

A.3 Variable Refactorings

AddInstanceVariable$(class, varName, initClass)$

$$
\begin{array}{rcl}
pre & : & \text{IsClass}(class) \\
& & \wedge\text{IsClass}(initClass) \\
& & \wedge\neg\text{HierarchyDefinesInstVar}(class, varName) \\
& & \wedge\neg\text{IsGlobal}(varName) \\
\hline
post & : & \text{InstanceVariablesDefinedBy}' = \\
& & \quad \text{InstanceVariablesDefinedBy}[class/\text{InstanceVariablesDefinedBy}(class) \cup \{varName\}] \\
& & \text{ClassOf}' = \text{ClassOf}[(class, varName)/\{initClass\}] \\
& & \text{IsExclusive}' = \text{IsExclusive}[(class, varName)/true] \\
& & \text{ReferencesToInstanceVariable}' = \text{ReferencesToInstanceVariable}[(class, varName)/\phi]
\end{array}
$$

RemoveInstanceVariable$(class, varName)$

$$\begin{array}{rcl} pre & : & \text{IsClass}(class) \\ & & \wedge\text{DefinesInstanceVariable}(class, varName) \\ & & \wedge\neg\text{HierarchyReferencesInstVar}(class, varName) \\ \hline post & : & \text{InstanceVariablesDefinedBy}' = \\ & & \quad \text{InstanceVariablesDefinedBy}[class/\text{InstanceVariablesDefinedBy}(class) - \{varName\}] \\ & & \text{IsExclusive}' = \text{IsExclusive}[(class, varName)/\perp] \\ & & \text{ReferencesToInstanceVariable}' = \text{ReferencesToInstanceVariable}[(class, varName)/\perp] \\ & & \text{ClassOf}' = \text{ClassOf}[(class, varName)/\perp] \end{array}$$

RenameInstanceVariable$(class, varName, newName)$

$$\begin{array}{rcl} pre & : & \text{IsClass}(class) \\ & & \wedge\text{DefinesInstanceVariable}(class, varName) \\ & & \wedge\neg\text{HierarchyDefinesInstVar}(class, newName) \\ & & \wedge\neg\text{IsGlobal}(newName) \\ \hline post & : & \text{InstanceVariablesDefinedBy}' = \\ & & \quad \text{InstanceVariablesDefinedBy}[class/ \\ & & \qquad (\text{InstanceVariablesDefinedBy}(class) - \{varName\}) \cup \{newName\}] \\ & & \text{IsExclusive}' = \\ & & \quad \text{IsExclusive}[(class, varName)/\perp][(class, newName)/\text{IsExclusive}(class, varName)] \\ & & \text{ReferencesToInstanceVariable}' = \text{ReferencesToInstanceVariable}[(class, varName)/\perp] \\ & & \quad [(class, newName)/\text{ReferencesToInstanceVariable}(class, varName)] \\ & & \text{ClassOf}' = \text{ClassOf}[(class, varName)/\perp][(class, newName)/\text{ClassOf}(class, varName)] \\ & & \forall(c, s) \in \text{ReferencesToInstanceVariable}(class, varName). \\ & & \text{Method}' = \text{Method}[(c, s)/\text{RenameReferences}(\text{Method}(c, s), varName, newName)] \end{array}$$

PullUpInstanceVariable$(class, varName)$

$$\begin{array}{rcl} pre & : & \text{IsClass}(class) \\ & & \wedge \exists c \in \text{Subclasses}(class).\text{DefinesInstanceVariable}(c, varName) \\ \hline post & : & \forall c \in \text{Subclasses}(class). \\ & & \text{InstanceVariablesDefinedBy}' = \\ & & \quad \text{InstanceVariablesDefinedBy}[class/\text{InstanceVariablesDefinedBy}(class) \cup \{varName\}] \\ & & \qquad [c/\text{InstanceVariablesDefinedBy}(c) - \{varName\}] \\ & & \text{IsExclusive}' = \text{IsExclusive}[(class, varName)/ \\ & & \quad \bigwedge_{cl \in \text{Subclasses}(class)} \text{IsExclusive}(cl, varName)][(c, varName)/\perp] \\ & & \text{ClassOf}' = \text{ClassOf}[(class, varName)/ \\ & & \qquad \bigcup_{cl \in \text{Subclasses}(class)} \text{ClassOf}(cl, varName)][(c, varName)/\perp] \\ & & \text{ReferencesToInstanceVariable}' = \text{ReferencesToInstanceVariable}[(class, varName)/ \\ & & \qquad \bigcup_{cl \in \text{Subclasses}(class)} \text{ReferencesToInstanceVariable}(cl, varName)] \\ & & \qquad [(c, varName)/\perp] \end{array}$$

PushDownInstanceVariable$(class, varName)$

$$
\begin{array}{rcl}
pre & : & \text{IsClass}(class) \\
& & \wedge \text{DefinesInstanceVariable}(class, varName) \\
& & \wedge \neg \text{ReferencesInstVar}(class, varName) \\
\hline
post & : & \forall c \in \text{Subclasses}(class). \\
& & \text{InstanceVariablesDefinedBy}' = \\
& & \quad \text{InstanceVariablesDefinedBy}[class/\text{InstanceVariablesDefinedBy}(class) - \{varName\}] \\
& & \qquad [c/\text{InstanceVariablesDefinedBy}(c) \cup \{varName\}] \\
& & \text{IsExclusive}' = \text{IsExclusive}[(class, varName)/\perp] \\
& & \qquad [(c, varName)/\text{IsExclusive}(class, varName)] \\
& & \text{ClassOf}' = \text{ClassOf}[(class, varName)/\perp][(c, varName)/\text{ClassOf}(class, varName)] \\
& & \text{ReferencesToInstanceVariable}' = \text{ReferencesToInstanceVariable} \\
& & \qquad\quad [(c, varName)/\text{ReferencesToInstanceVariable}(class, varName)] \\
& & \qquad\quad [(class, varName)/\perp]
\end{array}
$$

AbstractInstanceVariable$(class, varName, getter, setter)$

$$
\begin{aligned}
pre \quad : \quad & \text{IsClass}(class) \\
& \wedge \text{DefinesInstanceVariable}(class, varName) \\
& \wedge \text{DefinesSelector}(class, getter) \\
& \wedge \text{DefinesSelector}(class, setter) \\
& \wedge \text{IsGetter}(\text{Method}(class, getter), varName) \\
& \wedge \text{IsSetter}(\text{Method}(class, setter), varName)
\end{aligned}
$$

$$
\begin{aligned}
post \quad & \forall (c, s) \in \text{ReferencesToInstanceVariable}(class, varName), \\
& \qquad (c, s) \notin \{(class, getter), (class, setter)\}. \\
& \text{Method}' = \text{Method}[(c, s)/\text{AbstractVariable}(\text{Method}(c, s), varName, getter, setter)] \\
& \text{ReferencesToInstanceVariable}' = \\
& \qquad \text{ReferencesToInstanceVariable}[(class, varName)/\{(class, getter), (class, setter)\}]
\end{aligned}
$$

MoveInstanceVariable$(class, varName, destVarName)$

$$\begin{array}{rcl}
pre & : & \text{IsClass}(class) \\
& & \wedge\text{DefinesInstanceVariable}(class, varName) \\
& & \wedge\text{DefinesInstanceVariable}(class, destVarName) \\
& & \wedge\forall c \in \text{ClassOf}(class, destVarName).\neg\text{HierarchyDefinesInstVar}(c, varName) \\
& & \wedge\text{IsExclusive}(class, destVarName) \\
\hline
post & : & \forall c \in \text{ClassOf}(class, destVarName). \\
& & \text{InstanceVariablesDefinedBy}' = \\
& & \quad \text{InstanceVariablesDefinedBy}[class/\text{InstanceVariablesDefinedBy}(class) - \{varName\}] \\
& & \qquad [c/\text{InstanceVariablesDefinedBy}(c) \cup \{varName\}] \\
& & \text{IsExclusive}' = \text{IsExclusive}[(class, varName)/\perp] \\
& & \qquad [(c, varName)/\text{IsExclusive}(class, varName)] \\
& & \text{ClassOf}' = \text{ClassOf}[(class, varName)/\perp][(c, varName)/\text{ClassOf}(class, varName)] \\
& & \text{ReferencesToInstanceVariable}' = \text{ReferencesToInstanceVariable}[(class, varName)/\perp] \\
& & \qquad [(c, varName)/\{getter, setter\}] \\
& & \text{Method}' = \forall c \in \{class\} \cup \text{Subclasses}^*(class).\forall s. \\
& & \qquad \text{Method}[(class, s)/\text{AbstractVariable}(\text{Method}(class.s), varName, getter, setter)]
\end{array}$$

Appendix B

Analysis Functions

B.1 Primitive Analysis Functions

- IsClass($class$)
- IsGlobal($name$)
- ClassReferences($class$)
- Superclass($class$)
- Method($class, selector$)
- ReferencesToInstanceVariable($class, varName$)
- Senders($class, selector$)
- ClassOf($class, varName$)
- InstanceVariablesDefinedBy($class$)
- IsExclusive($class, varName$)

B.2 Derived Analysis Functions

$$\text{Subclasses}(\mathit{class}) \equiv \{c | \text{Superclass}(c) = \mathit{class}\}$$

$$\text{Subclasses}^*(\mathit{class}) \equiv \begin{cases} \phi & \text{if Subclasses}(\mathit{class}) = \phi \\ \bigcup_{c \in \text{Subclasses}(\mathit{class})} \{c\} \cup \text{Subclasses}^*(c) & \text{otherwise} \end{cases}$$

$$\text{IsEmptyClass}(\mathit{class}) \equiv \neg \exists s.\text{DefinesSelector}(\mathit{class}, s) \wedge \neg \exists v.\text{DefinesInstanceVariable}(\mathit{class}, v)$$

$$\text{Superclass}^*(\mathit{class}) \equiv \begin{cases} \phi & \text{if Superclass}(\mathit{class}) = \perp \\ \{\text{Superclass}(\mathit{class})\} \cup \text{Superclass}^*(\text{Superclass}(\mathit{class})) & \text{otherwise} \end{cases}$$

$$\text{DefinesSelector}(\mathit{class}, \mathit{selector}) \equiv \text{Method}(\mathit{class}, \mathit{selector}) \neq \perp$$

$$\text{LookedUpMethod}(\mathit{class}, \mathit{selector})$$

$$\equiv \begin{cases} \perp & \text{if } \mathit{class} = \perp \\ \text{Method}(\mathit{class}, \mathit{selector}) & \text{if Method}(\mathit{class}, \mathit{selector}) \neq \perp \\ \text{LookedUpMethod}(\text{Superclass}(\mathit{class}), \mathit{selector}) & \text{otherwise} \end{cases}$$

$$\text{UnderstandsSelector}(\mathit{class}, \mathit{selector}) \equiv$$

$$\exists c \in \{\mathit{class}\} \cup \text{Superclass}^*(\mathit{class}).\text{DefinesSelector}(c, \mathit{selector})$$

$$\text{DefinesInstanceVariable}(\mathit{class}, \mathit{varName}) \equiv \mathit{varName} \in \text{InstanceVariablesDefinedBy}(\mathit{class})$$

$$\text{HierarchyDefinesInstVar}(\mathit{class}, \mathit{varName}) \equiv$$

$$\exists c \in (\{\mathit{class}\} \cup \text{Superclass}^*(\mathit{class}) \cup \text{Subclasses}^*(\mathit{class})).$$

$$\text{DefinesInstanceVariable}(c, \mathit{varName})$$

$$\text{HierarchyReferencesInstVar}(\mathit{class}, \mathit{varName}) \equiv$$

$$\exists c \in (\{\mathit{class}\} \cup \text{Superclass}^*(\mathit{class}) \cup \text{Subclasses}^*(\mathit{class})).$$

$$\text{ReferencesToInstanceVariable}(c, \mathit{varName}) \neq \phi$$

Appendix C

Abstract Syntax Tree Functions

- IsGetter($method$)
- IsSetter($method$)
- AbstractVariable($method$)
- MovedMethod($method$)
- ForwarderMethod($method$)
- UnboundVariables($method$)
- RenameReferences($method, oldName, newName$)

References

[ABW98] Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The Design Patterns Smalltalk Companion*. Addison Wesley, 1998.

[ACS84] James Archer, Richard Conway, and Fred Schneider. User recovery and reversal in interactive systems. *ACM Transactions on Programming Languages and Systems*, 6(1):1–19, January 1984.

[AIS+79] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobsen, Ingred Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Pres, New York, 1979.

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techiniques, and Tools*. Addison-Wesley, 1988.

[BD77] R.B. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[Bec96] Kent Beck. *Smalltalk Best Practices Patterns, Vol. 1: Coding*. Prentice-Hall, 1996.

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[Ber91] Paul L. Bergstein. Object-preserving class transformations. In *Proceedings of OOPSLA '91*, 1991.

[BFJR98] John Brant, Brian Foote, Ralph Johnson, and Don Roberts. Wrappers to the rescue. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, July 1998.

[BG79] R. Balzer and N. Goodman. Principles of good software specification. In *Proc. on Specifications of Reliable Software*, pages 58–67. IEEE, 1979.

[BG94] R.W. Bowdidge and W.G. Griswold. Automated support for encapsulating abstract data types. In *ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 97–110, December 1994.

[BG97] Robert Bowdidge and William Griswold. How software engineering tools organize behavior during the task of data encapsulation. *Empirical Software Engineering*, 2(3):221–268, 1997.

[BG98] R.W. Bowdidge and W.G. Griswold. Supporting the restructuring of data abstractions through manipulation of a program visualization. *Transaction on Software Engineering and Methodology*, April 1998.

[BL71] L.A. Belady and M.M. Lehman. Programming system dynamics or the metadynamics or systems in maintenance and growth. IBM Research Report RC3546, IBM, 1971.

[BL76] L.A. Belady and M.M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3), 1976.

[BMR$^+$96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley & Sons, 1996.

[Boe75] B.W. Boehm. The high cost of software. In E. Horowitz, editor, *Practical Strategies ofr Developing Large Software Systems.* Addison-Wesley, Reading, MA, 1975.

[Bow95] R.W. Bowdidge. *Supporting the Restructuring of Data Abstractions through Manipulation of a Program Visualization.* PhD thesis, University of California, San Diego, Department of Computer Science and Engineering, 1995.

[Bro75] Frederick P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, 1975.

[Bro80] P.J. Brown. Why does software die? Pergamon infotech state of the art report, Pergamon Infotech Ltd., 1980. Republished in Arnold, Robert S., Editor, Tutorial on Software Restructuring. IEEE Press, 1986.

[Cas91] Eduardo Casais. *Managing Evolution in Object Oriented Environments: An Algorithmic Approach.* PhD thesis, University of Geneva, 1991.

[CD93] Charles Consel and Olivier Danvy. Tutorial on partial evaluation. Presented at POPL '93, 1993.

[Deu89] L. Peter Deutsch. *Software Reusability - Volume II: Application and Experience*, chapter Design Reuse and frameworks in the Smalltalk-80 System, pages 57–72. LOOKUP, 1989.

[DR93] Nachum Dershowitz and Uday Reddy. Deductive and inductive synthesis of equational programs. *Journal of Symbolic Computation*, 11, 1993.

[Fea82] Martin Feather. A system for assisting program transformation. *ACM Transactions on Programming Languages and Systems*, 4(1):1–20, January 1982.

[FOW87] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Language Systems*, 9(3):319–349, July 1987.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Exisiting Programs.* Addison-Wesley, 1999.

[GHH97] Richard Garzaniti, Jim Huangs, and Chet Hendrickson. Everything i need to know i learned from the chrysler payroll project. In *Conference Addendum to the Proceedings of OOPSLA '97*, 1997.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissedes. *Design Patterns: Elements of Object-Oriented Style*. Addison Wesley, 1995.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[GL91] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Trans. Software Eng.*, 17(8):751–761, August 1991.

[GN93] William G. Griswold and David Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228–269, July 1993.

[Gri91] William G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, University of Washington, August 1991.

[Gri93] W.G. Griswold. Direct update of dataflow representations for a meaning-preserving program restructuring tool. In *ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering, Software Engineering Notes*, volume 18, pages 42–55, December 1993.

[Hew77] Carl Hewitt. Control structures as patterns of message passing. *Artificial Intelligence*, 8:323–363, 1977.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[JBR95] Ralph Johnson, John Brant, and Don Roberts. The design of a refactoring tool. Presented at ICSE-17 Workshop on Program Transformation for Software Evolution, Seattle, WA., April 1995.

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, June/July 1988.

[KB70] D. Knuth and P. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Oxford: Pergamon Press, 1970.

[KKL$^+$81] D.J. Kuck, R.H Kuhn, B. Leasure, D.A. Padua, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[Kor98] Walter Fred Korman. Elbereth: Tool support for refactoring java programs. Master's thesis, University of California, San Diego, 1998.

[Leh78] M.M. Lehman. Why software projects fail. In *Infotech State of the Art Conference*. Pergamon Press, 1978.

[LHR88] K. Lieberherr, I. Holland, and A. Riel. Object-oriented programming: An objective sense of style. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1988.

[LS87] Jacques Loeckx and Kurt Sieber. *The Foundations of Program Verification.* John Wiley & Sons, 1987.

[Mar95] Brian Marick. *The Craft of Software Testing: Subystem Testing Including Object-Based and Object-Oriented Testing.* Prentice Hall, 1995.

[Moo96] Ivan Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *Proceedings of OOPSLA '96*, pages 235–250, 1996.

[Mor97] John David Morgenthaler. *Static Analysis for a Software Transformation Tool.* PhD thesis, University of California, San Diego, 1997.

[Obj98] ObjectShare, Inc. Visualworks 3.0 base image, 1998.

[OJ90] William F. Opdyke and Ralph E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPA)*, September 1990.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois, 1992. Available as Technical Report No. UIUCDCS–R–92–1759. Postscript: /pub/papers/refactoring/opdyke-thesis.ps.Z on st.cs.uiuc.edu.

[OR95] William F. Opdyke and Don Roberts. Refactoring. Tutorial presented at OOPSLA '95: 11th Annual Conference on Object-Oriented Programming Systems, Languages and Applications, Austin, Texas, October 1995.

[Par72] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Pre87] Roger S. Pressman, Ph.D. *Software Engineering: A Practitioners Approach.* McGraw-Hill, second edition, 1987.

[RBJ97] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, 3(4), 1997.

[Ric53] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. AMS*, 89, 1953.

[RJ98] Don Roberts and Ralph Johnson. *Pattern Languages of Program Design 3*, chapter 26 - Patterns for Evolving Frameworks. Addison-Wesley, 1998.

[Sch81] W. Scherlis. Program improvement by internal specialization. In *ACM POPL*, 1981.

[Sch86] W. Scherlis. *Advanced Programming Environments*, chapter Abstract Data Types, Specialization and Program Reuse. Springer, 1986.

[Sch94] W. Scherlis. Boundary and path manipulations on abstract data types. In *IFIP 94*, 1994.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1987.

[Wei79] M. Weiser. *Program Slicing: Formal, Psychological and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, 1979.

[Wei82] M. Weiser. Programmers use slices when debugging. *Comunications of the ACM*, 25(7):446–452, 1982.

[Wei83] Gerald Weinberg. Kill that code! *Infosystems*, pages 48–49, Aug. 1983.

[Wei84] M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10:352–357, July 1984.

Vita

Donald Bradley Roberts was born on April 22nd, 1968 in St. Louis, Missouri. He attended the University of Illinois at Urbana-Champaign, where he earned a bachelor of science degree in computer science in 1990. During the course of his studies, he was awarded the James Snyder Award for outstanding sophomore in computer science, the Daniel Slotnick Award for outstanding senior in computer science, the NCR award for excellence in computer science, the Tandem Computer award for excellence in computer science, and the Bronze Tablet award for finishing in the top 3% of all graduates.

Dr. Roberts immediately entered the graduate program at the University of Illinois at Urbana-Champaign where he was a University Fellow for one year and a Hertz Fellow for five years. He obtained his master of science degree in computer science in 1992 with his thesis entitled *Register Allocation for Nonhomogeneous Architectures.*

Dr. Roberts pursued his doctoral research in the area of refactoring object-oriented programs under Dr. Ralph Johnson. During the course of his research, he also worked as a consultant on several industry projects, exposing him to the need for refactoring tools in this arena, along with the constraints that such an environment enforces. He received his Ph.D. from the University of Illinois in 1999 with his dissertation entitled *Practical Analysis for Refactoring.*