REFACTORING OBJECT-ORIENTED FRAMEWORKS

BY

WILLIAM F. OPDYKE

B.S., Drexel University, 1979
B.S., Drexel University, 1979
M.S., University of Wisconsin - Madison, 1982

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, 1992

Urbana, Illinois

REFACTORING OBJECT-ORIENTED FRAMEWORKS

William F. Opdyke, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1992
Ralph E. Johnson, Advisor

This thesis defines a set of program restructuring operations (refactorings) that support the design, evolution and reuse of object-oriented application frameworks.

The focus of the thesis is on automating the refactorings in a way that preserves the behavior of a program. The refactorings are defined to be behavior preserving, provided that their preconditions are met. Most of the refactorings are simple to implement and it is almost trivial to show that they are behavior preserving. However, for a few refactorings, one or more of their preconditions are in general undecidable. Fortunately, for some cases it can be determined whether these refactorings can be applied safely.

Three of the most complex refactorings are defined in detail: generalizing the inheritance hierarchy, specializing the inheritance hierarchy and using aggregations to model the relationships among classes. These operations are decomposed into more primitive parts, and the power of these operations is discussed from the perspectives of automatability and usefulness in supporting design.

Two design constraints needed in refactoring are class invariants and exclusive components. These constraints are needed to ensure that behavior is preserved across some refactorings. This thesis gives some conservative algorithms for determining whether a program satisfies these constraints, and describes how to use this design information to refactor a program.

To Brenda, for her encouragement and support.

Acknowledgements

I thank my advisor, Prof. Ralph Johnson, for his support, guidance and patience during my studies at the University of Illinois. I learned much from our weekly Monday working lunches and from other feedback he gave me. I was amazed by both his insights and his stamina. He invested his most valuable resource on my behalf: his time.

I also thank the other members of my committee. Prof. Simon Kaplan was helpful in pointing out places in several drafts of the thesis where clarity could be improved and claims made more precise. His views on supporting the software design process fundamentally influenced the approach I took in my work. Prof. Sam Kamin provided valuable insights regarding data flow analysis. I also thank Prof. Geneva Belford and Prof. Larry Jones for their input.

I owe thanks to many others who were at the University of Illinois during my graduate program. Prof. Mehdi Harandi and Prof. Roy Campbell provided helpful guidance early in my graduate studies. During my early research, Peter Madany provided invaluable input from his work on the *Choices* file system framework. Vince Russo was helpful in pointing me toward Peter. Mick Murphy prototyped a front end to my refactoring prototype. Steve March provided expert help in machine administration of my workstation.

Brian Foote was helpful at several stages of my research. Early on, he shared with me what it was like to work with Ralph Johnson. Later, he was a good sounding board for ideas. Throughout, he shared my liking for mushroom pizza.

I also owe thanks to several current and former members of AT&T Bell Laboratories. Paul Zislis, my former department head at Bell Labs, was instrumental in my being selected for AT&T's full-time doctoral support program. I hold in highest regard his personal integrity and concern for his staff. Warren Montgomery served at various times as supervisor and mentor. Early on, he provided helpful advice as I considered several possible research topics. Later, he provided words of encouragement during difficult stages of my research. Moody Ahmad, John Degan, Moe Grzelakowski and Jack Wang also provided management support for my work.

I thank Bjarne Stroustrup for his encouragement of my pursuing this area of research. Jim Vandendorpe, Dewayne Perry and Jim Coplien reviewed parts of my thesis. Larry Mayka provided expert advice in using Common LISP for my prototyping.

At Drexel University, Prof. James Maginnis sparked my interest in many areas of computer science. At the University of Wisconsin, Prof. Randy Katz sparked my interest in computer science research.

I thank those many others not named here who provided encouragement and assistance during my graduate studies.

I thank my parents for their support. I especially appreciated my father's words of encouragement, and his ability to put today's 'crises' in perspective. I thank my Aunt Nancy for being an example to me.

I am especially thankful to my wife Brenda and our children. When I began my doctoral program, I was 32 years old, married, and the father of several preschool aged children. The transition to graduate school was an adjustment for all of us.

I thank my children David, Andrew, Jamie and Daniel for many encouraging times together, and for loving me more for who I am than for what I've done.

I am especially thankful to my wife Brenda for her support during my graduate studies. Without her, I never would have made it through the program. As a wife and mother, she picked up the slack and encouraged me in an extraordinary way. She deserves an award at least as valuable as my PhD.

Finally, I am thankful to God for having granted me the skills and opportunities that made this possible.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 The Problem

Design is hard. The design of reusable software is especially hard. Reusable software usually is the result of many design iterations. Some of these iterations occur after the software has been reused, and the resulting changes affect not only the design of the reusable software, but also the design of other software that is using it. Making software easier to change makes subsequent design iterations easier, and makes the software more reusable.

Although there have been no scientific studies that validate the claim, it is nonetheless a strongly held conclusion among many practitioners that object-oriented software is easier to change than conventional software [79]. Some changes to object-oriented software can be made simply by adding new subclasses or by adding new operations on existing classes, while leaving most of the original software unchanged.

However, object-oriented software is harder to change than it might at first appear to be. Changing an object-oriented system often requires changing the abstractions embodied in *existing* object classes and the relationships among those classes. This involves structural changes such as moving variables and functions between classes and partitioning a complicated class into several classes. When a structural change is made to a class or set of classes, corresponding changes may also be needed elsewhere in a program, due to naming, typing and scoping (inheritance) dependencies. Tracking down the dependencies by hand and consistently updating the program can be time consuming, difficult and error prone.

The reusability benefits of object-oriented programming can be difficult to realize without some form of automated support for making these structural changes [59, 98].

1.2 A Proposed Solution

This dissertation describes an approach for providing automated support for the restructuring of object-oriented programs.

People usually think about software changes either at a high level, in terms of features to be added to a system, or at a low level, in terms of lines of code to be changed. *Refactorings* are reorganization plans that support change at an intermediate level. Consider, for example, the refactoring that moves a member function from one class to another. The reason for applying this refactoring might be to support a new feature in the system, which will be implemented using a class for which the member function is needed. This refactoring maps into low level changes not only in these two classes but possibly also in other classes that invoke the member function.

Refactorings do not change the behavior of a program; that is, if the program is called twice (before and after a refactoring) with the same set of inputs, the resulting set of output values will be the same. Refactorings are behavior preserving so that, when their preconditions are met, they do not "break" the program.

While refactorings do not change the behavior of a program, they can support software design and evolution by restructuring a program in the way that allows other changes to be made more easily. Complicated changes to a program can require both refactorings and additions. For example, a new feature might be added to a program by first refactoring part of a complex class into a component class, and then using the component class in defining the new feature.

There are several cases where refactorings might be applied [85]:

1. *Extracting a reusable component.* For example, an industrial process control system had served user needs well for several years. Customers require a new product to support new process approaches, but with a user interface that is compatible with the older system. When the older system was originally designed and implemented, it was factored such that the user interface functions were interwoven with other, obsolete functions. Extracting the user interface component first requires refactoring the existing software.

2. *Improving consistency among components.* For example, two components of a software system are implemented by different project members. While these components were initially thought to be distinct, it was later discovered that they shared a common abstraction. To make the design of the system easier to understand, and to reduce future maintenance costs, it is desirable to refactor the system to make more explicit the commonalities between the two components.

3. *Supporting the iterative design of an Object-Oriented Application Framework.* An object-oriented application framework is an abstract design of an application, consisting of an abstract class for each major component. It is an important object-oriented technique to facilitate design-level reuse [127]. Good frameworks are usually the result of many design iterations and a lot of hard work, involving structural changes.

   For example, the application framework for managing files in an operating system [77] began as an implementation of only one file format. Then, it was to be extended to handle additional file formats. Unfortunately, in the initial implementation the more general, common abstractions were intertwined with features specific to the file format supported. Refactoring was necessary to separate out the common abstractions in order to improve the design of the framework and facilitate reuse.

The research described here focused on the third area; namely, in how these refactorings can support the iterative design of object-oriented application frameworks. The other two areas are covered as well, since as a framework matures reusable components are extracted and consistency among components is improved.

There are several reasons why this research has focuses on refactoring *object-oriented* systems:

1. Compared with more traditional software development approaches, object-oriented programming makes refactoring *more feasible* by making more explicit the structural information needed to refactor a program.

2. Refactoring is *especially important* in object-oriented programming. Some in the object-oriented programming culture have placed high value in designing and redesigning software to make it more reusable [59, 98]. In some cases, the best way to improve the design of a program is to re-write it. In other cases, restructuring a program (rather than rewriting it) may be a more practical way to improve its design.

1.2.1 Dealing With the Complexities of Refactoring

There are several aspects that make refactoring object-oriented frameworks difficult:

1. There is no theory (i.e. systematically organized explanation) of how people refactor object-oriented application frameworks and the kinds of refactorings they make.

2. Some refactoring operations require design insights that are hard to obtain by inspecting the program to be refactored.

3. Two (conflicting) objectives of a set of refactorings are that they be both behavior preserving and expressive. They should be powerful enough to handle the important program restructuring tasks at a level that shields the user from most of the complexity. At the same time, they should be restrictive enough to preserve the behavior of the program. In shielding complexity from the user, the refactorings themselves become more complex. These complexities, and the limitations in inferring design intent, make it difficult to ensure that the intended behavior is preserved.

A tool that provides automated support for refactoring should guarantee that its operations preserve the behavior of a program. One of the reasons why programs are not refactored is that any change to the program runs the risk of introducing defects into the program. An important focus of this thesis is on how to tell whether a refactoring can be applied to a program without changing the program's behavior. A major motivation for automating refactoring is to ensure that defects are not introduced into a program.

A refactoring that can be applied safely to a program will not necessarily improve its design. On the contrary, applying arbitrary refactorings to a program is more likely to corrupt the design rather than improve it, even though the behavior of the program is preserved. A refactoring improves design if the resultant code units correspond to meaningful abstractions that make it easier to refine or extend the program. What abstractions are meaningful depends on the application and on the designer. This implies that refactoring tasks, especially the most complex tasks, require some interaction with the designer. A refactoring tool can help a designer by providing the right set of refactorings, and by ensuring that each refactoring is applied correctly, but it cannot decide which refactorings to apply. Thus, refactoring cannot be completely automated.

To understand how people refactor programs, related research [9, 30, 59, 71, 98] was surveyed, and an analysis was done of the structural changes made to the *Choices* file system framework [77] over a two year period. The *Choices* file system framework was developed using the *C++* programming language [114]. Based on that analysis, the following set of eight basic refactorings was compiled [85]:[1]

1. defining an abstract superclass of one or more existing classes
2. specializing a class by defining subclasses, and using subclassing to eliminate conditional tests
3. changing how the whole/part association between classes is modeled, from using inheritance to using an instance hierarchy of aggregates and their components
4. moving a class within and among inheritance hierarchies
5. moving member variables and functions
6. replacing a code segment with a function call
7. changing the names of classes, variables and functions
8. replacing unrestricted access to member variables with a more abstract interface

To someone interested in adding new features to an object-oriented program, some of these refactorings may seem low level and not particularly difficult; however, these refactorings require attention to many details regarding the interrelationships of parts of a program. Some of these refactorings employ other, more primitive refactorings. For example, defining an abstract superclass may involve changing the name of members (variables and functions), splitting functions and migrating common members to the superclass. In order to better deal with the complexity of the refactoring process, the refactorings were organized into:

- a set of twenty six *low-level* refactorings, such as renaming a member variable. These refactorings are defined in chapter five.
- a set of three more abstract, *high-level* refactorings: defining an abstract superclass, simplifying conditional statements with subclassing, and several component-related refactorings. These refactorings are detailed in chapters six through eight.

The low-level refactorings are simple enough that in most cases it is trivial to show that they are behavior-preserving. While most of these low-level operations have little theoretical difficulty, their implementation can be complicated. The high-level refactorings are more expressive and theoretically interesting refactorings. They are constructed using the low-level refactorings.

1.2.2 Representing Design Intent

Some of the more powerful refactorings require an explicit representation of design intent. For example, if a class is to be specialized by defining new subclasses, deciding what subclasses should be added depends on the meaningful specializations for that application. Two kinds of design information are needed for some of the high-level refactorings:

[1]The refactorings are described using C++ terms such as *member variable* and *member function*, which correspond to the Smalltalk-80™ [49] terms *instance variable* and *method*. Smalltalk-80 is a trademark of ParcPlace Systems.

1. a *class invariant*, which is a predicate, defined on the values of member variables, that is true for the lifetime of an instance of the class. Chapter seven defines refactorings for simplifying a class by defining subclasses. In member functions of the newly created subclasses, class invariants are used in simplifying conditional statements.

2. a *list of component member variables*, which is a subset of the list of member variables defined in a class. Component member variables model a special *is-part-of* relationship with the class that contains them, referred to as an *aggregate* class. Chapter eight defines refactorings that involve components and the aggregate classes that contain them. Some of these refactorings require that the components be *exclusive*, that is, that they not be shared among multiple aggregate objects.

This design intent is not made explicit (or enforced) in a language such as C++; thus, a system for supporting refactorings must be able to certify class invariants and lists of components. In general, it is undecidable whether a predicate is a class invariant or whether a member variable is a component. Since refactorings must be behavior preserving, it is better to mistakenly decide that a predicate is not a class invariant (and so not be able to perform a legal refactoring) than it is to mistakenly decide that a predicate is a class invariant and eventually perform an illegal refactoring. Thus, a system for supporting refactorings will use conservative algorithms for deciding whether a predicate is a class invariant or a member variable is a component. The thesis will describe some conservative algorithms for both problems that are based on data-flow analysis.

1.3 Contributions

The major contributions of this research are:

1. It identifies a set of program restructurings (refactorings) that people apply to object-oriented application frameworks.

2. It shows how to automatically support refactorings in a way that preserves the behavior of a program.

3. It defines in detail three of the most complex refactorings.

4. It defines design constraints needed in refactoring, specifically class invariants and exclusive components.

The research described here could serve as the basis for a *software refactory* [85]. Whereas past research into a *software factory* (for example, [110]) has focused on generating a software program from specifications, this research has focused on how to restructure an existing program to make it easier to understand, change and reuse. Before a practical software refactory can be realized, however, several additional research issues need to be addressed, as noted in chapter eleven.

The next two chapters present the motivation for restructuring object-oriented systems, and describe the refactoring approach. Chapter four describes several issues related to preserving behavior during refactoring. Chapter five describes the low-level refactorings. Chapters six through eight describe the three highest level refactorings, which are the most expressive and most algorithmically complex. The final three chapters provide extended examples, survey closely related work, briefly discuss an implementation of the approach and present conclusions.

Chapter 2

Motivation

A software project "is capable of becoming a monster of missed schedules, blown budgets and flawed products" [24]. One approach to achieving meaningful reductions in software costs is to acquire an existing software system rather than developing a new one. Often, however, the available software systems do not provide an exact fit for the problem at hand. Software that solves a *similar* problem might be available, but such software may need to be changed in some way before it can be reused. These changes may involve restructuring the software.

Object-oriented programming is often touted as promoting software reuse [45]. Sometimes however the benefits of the object-oriented approach are overstated, and claims are made that features can be added to an object-oriented system without disturbing the existing implementation. As this chapter will show, object-oriented software often needs to be restructured before it can be reused.

2.1 Background: Software Reuse

The high costs of developing software motivate the reuse and evolution of existing software. Software reuse in its broadest sense involves reapplying knowledge about one software system to reduce the efforts of developing and maintaining another system. Closely related to software reuse is software maintenance, where knowledge about a software system is used to develop a version that refines or extends it.[1]

Approaches that support reuse address one or more of the following four important aspects [19]:

1. *finding* a reusable component. This is usually is not as simple as finding an exact match, but rather involves finding the most similar component.

2. *understanding* the component. Understanding what a component does is important in order to use it, but developing that mental model is difficult [24, 88].

[1] Most (60%) of the activities analyzed in Osborne's software maintenance study [80], were found neither to be corrective (that is, diagnosing and fixing errors) nor adaptive (changing software to work with new hardware and peripherals) but rather were *perfective* activities, where pressure was brought to bear on developers to extend and enhance the functionality of a system. Perfective software maintenance is closely related to software reuse.

3. *modifying* a component or a set of components. Understanding what changes are needed has proven to be human-intensive and few tools have proven very helpful. As will be shown below, these modifications often involve restructuring.

4. *composing* the components together. The composition process can be difficult, especially when a component has the dual purposes of being a useful independent entity and being used to create other composite structures.

While some software reuse techniques have focused at the code level, others have focused on design-level reuse. There are limitations on the reuse of code: it works best when the domain is narrow and well understood and the underlying technology is very static. Sometimes the design of software is reusable even when the code is not. However, a major problem with design-level reuse is that there is no well-defined representation system for design.

Reuse does not happen by accident; one needs to plan to reuse software and look for software to reuse. Reuse requires the right attitude, tools and techniques [59, 117]. Tools and techniques to support software reuse include compositional and generational approaches [18]. The composition-based model of reuse is based on the notion of plugging components together, with little or no modification of those components, in order to create target software systems. The components might be code skeletons [32, 68], subroutines [32, 68, 96] or functions. The generation-based approach, on the other hand, is aimed at reusing patterns that drive the creation of specific or customized versions of themselves. Application generators [83] and some program transformation systems [23, 33, 43, 83] are examples of generation-based systems.

Software restructuring relates to each of the four important aspects of reuse listed above. Restructuring a program can make it easier to *understand* the design of a program and can assist in *finding* reusable components. Some restructurings *modify* a component to make it more reusable; such components can be easier to *compose* together for an application. The following section discusses software restructuring.

2.2 Background: Software Restructuring

Software sometimes needs to be restructured before it is reused. Arnold [3] defines software restructuring as "the modification of software to make the software (1) easier to understand and to change or (2) less susceptible to error when future changes are made."

A major goal of software restructuring is to preserve or increase the value of a piece of software. Restructuring a system may make it possible to add more features to the existing system, or make the software more reusable in other systems. Software restructuring approaches have become increasingly attractive as the cost of programmer time relative to computer time has increased. Software restructuring is most often used during software maintenance, where the lack of software structure often is most evident and expensive. However, it can also be applied in the earlier design and development phases.

Software structure can be broadly defined as a collection of software attributes that make sense to the perceiver [3]. Since programmers' perceptions differ from each other, and a programmer's perception can change over time, the notion of software structure is dynamic.

Many factors can contribute to poor software structure. These factors include an inadequate design methodology [14], absence of development and maintenance standards [2], buggy optimizations [113] and expedient but poorly conceived changes made to the software system to reflect changes in the environment in which it operates [25].

Weinberg in [123] suggests that very small changes to a software system are much more prone to error than larger changes, because people tend to take very small changes less seriously and are therefore less likely to test them adequately. The lower level refactorings described in this thesis are examples of small changes that if not done carefully can lead to errors in a program.

Improving a programmer's perception of the structure of a software system doesn't necessarily require modifying the code. Design recovery is a software reengineering approach that recreates design abstractions from a collection of code, existing documentation, general knowledge of the domain and heuristic reasoning [17, 34]. Design recovery has been proposed using program structures (*cliches*) and graph parsing [97], a heuristic-based concept-recognition mechanism with multiple views of program knowledge [55], and an approach using an intermediate modular interconnection language to generate design descriptions [35]. None of these approaches involve code changes. Also, they do not directly address how to change the implementation once the existing structure is understood; although clearly the knowledge gathered during software reengineering should help a developer determine where changes need to be made.

Many other software restructuring approaches involve code changes. A major purpose of these changes is to infuse the code with structure, making the flow of control in a program more explicit. Approaches have been proposed based on code inspections and walk throughs [42, 47], adherence to software metrics [11], maintainability measures [26] or other criteria [29, 52, 65, 68, 78, 87, 88, 90, 112]. Many of the earliest commercial products in this area were developed for restructuring COBOL programs [75, 82, 121].

Several techniques have been developed based on structured programming guidelines; these include goto elimination [4, 20, 128], case statement refinement [73] and other techniques (e.g., [29]). Parnas [87] gives principles for partitioning a system into modules to increase its maintainability. Lyons [75] and Morgan [82] describe tools that reduce gotos, remove dead (unreachable) code, convert notes to comments, physically group I/O, and highlight looping conditions. Other tools to support software restructuring include pretty printers and code formatters, integrated programming environments [10, 122] and rule-based program transformation systems [91].

Software restructuring continues to be an important area of software engineering research.

2.3 Object-Oriented Programming, Reuse and Restructuring

Object-oriented programming is often touted as promoting software reuse [45]. However, it is not a panacea. This section discusses some of the strengths of object-oriented programming regarding reuse, and some of its limitations.

2.3.1 Features of Object-Oriented Systems That Support Reuse

Most object-oriented languages combine several features that support reuse: data abstraction, class inheritance and polymorphism.

Most object-oriented languages provide *data abstraction* using classes. Class definitions provide encapsulation and in some cases provide information hiding, and can often serve as reusable components.

Class inheritance allows the operations and possibly the internal structure of a class to be inherited and reused by its subclasses. Inheritance is sometimes used to represent specialization hierarchies, where the subclasses are specializations of the superclass. Another common use of inheritance is to support programming by difference; one way to reuse the existing software is to create a subclass and represent the differences in the newly defined subclass.

Polymorphism is another powerful technique of object-oriented languages. Polymorphism is the notion that a procedure can be invoked on an object without knowing its exact type. An example of a polymorphic operation is an operation to find the maximal element in an array. The operation could be defined in the array class as a series of pairwise comparisons on the elements in the array. In an object-oriented system, this could be implemented as a series of calls to array elements to compare themselves with another element and return the maximal element. The operation could be defined once on the array class and then reused on arrays of different types, as long as each type of array element understood how to compare itself with another array element of that type.

This combination of features is supplemented in Smalltalk and some other object-oriented languages with a sophisticated user interface and browsing facilities that help make explicit the inheritance relationships among classes and the protocol (set of operations) supported by each class. These features encourage the identifying and reusing of existing code rather than writing new code from scratch.

2.3.2 Object-Oriented Application Frameworks and Reuse

While object-oriented programming makes program components more reusable, in the long run reusing the design of an application is more important than reusing the implementation of any one of its components. Biggerstaff and Richter note that the fundamental problem preventing successful design level reuse is finding the right representation of design. Such a representation should [16]:

- represent knowledge about implementation structures in a factored form
- permit partial specifications that can be incrementally extended
- allow flexible couplings between a design and its various interpretations
- support degrees of abstraction and precision.

An important object-oriented technique for facilitate design-level reuse is an *application framework.* An application framework represents the design of an application; it consists of a set of classes (many of which may be abstract), each representing a major component. Frameworks capture the designs of interfaces and the way functionality is divided among components, which Deutsch argues are the key intellectual content of software [40]. Frameworks define an *external interface* that is constant across all uses of a framework, and an *internal interface* that is a set of constraints on subclass code or a component object's protocol.

A framework is a mixture of abstract and concrete classes. One of the main characteristics of a framework is that it is designed to be refined. It can be refined by changing the configuration of its components or by creating new kinds of components (i.e. new classes, often as subclasses of existing classes). A mature framework will have a large class library of concrete subclasses of each abstract class, so that most of the time an application can be plugged together from existing components. Even when new subclasses are needed, they are easy to produce; their abstract superclass provides their design and much of their code, and the already existing concrete subclasses provide examples of how to subclass from the abstract superclass.

Thus, an object-oriented application framework is a representation that supports design level reuse by representing knowledge in a factored form, permitting partial specifications, and supporting degrees of abstraction. Furthermore, subclassing can be used in an application framework to model various concrete representations of a common abstraction.

2.3.3 Reusing & Refactoring Object-Oriented Software

One might be tempted to conclude that an object-oriented system, once developed, can be reused or extended simply by combining components of existing classes in different ways, by adding operations to existing classes or by subclassing from existing classes. Often, however, object-oriented software cannot be reused without first being restructuring. There are several reasons for this [31, 59]:

1. When developing a software application, it is difficult to determine *a priori* what classes embody the important concepts for that application and how they interrelate. Experience has shown that a useful taxonomy of classes is discovered through an iterative process of exploration. As an understanding of the application improves, the system often needs to be restructured and the abstractions embodied in existing classes often need to be changed.

2. Even after a system has matured through several iterations, sweeping structural changes might still be necessary. The software system must operate in an environment that is constantly changing, and the software system must satisfy user needs that are constantly changing as well.

3. When attempts are made to reuse software across projects, new issues arise. A system may need to be partitioned differently, due to organizational and other factors, in order for a new project to reuse it. Thus, some restructuring may be needed to effect reuse.

4. Subclassing in an object-oriented system can simultaneously serve several purposes, like code sharing, type validation and modeling generalization/specialization relationships. It can be difficult to reconcile all of these purposes without some restructuring when a change is proposed.

None of these issues is exclusive to object-oriented systems. The fourth issue applies to languages that have subtypes; the other three issues apply more generally. The important point here is that these issues do not disappear when object-oriented technology is introduced. One of the essential difficulties in developing software is that the software must conform to human institutions and systems that interface with it, and that those forces are constantly changing [24].

Nonetheless, the motivation to restructure object-oriented programs is somewhat different from the motivation for many software restructuring approaches in the past. Many of the software restructuring techniques described earlier were motivated by the need to understand and evolve unstructured or poorly structured programs. However, with object-oriented systems, some structural information is already explicit within the class and inheritance structures; the integrated development environments for Smalltalk and other object-oriented languages make understanding a program structure easier by providing facilities for abstracting and browsing the program. With object-oriented systems refactoring is needed not so much to infuse structure into a poorly structured program, but rather to refine the design of an already structured program, and make it easier to reuse.

The need to investigate approaches for software restructuring has been motivated by several sources within the object-oriented community. These include work in object-oriented databases, and research into improving the style of object-oriented programs to increase their reusability. These efforts are briefly discussed below and further detailed in chapter ten.

Object-oriented databases require structural changes for many of the same reasons that object-oriented programs do: users often need to view the database schema (structure) differently, and the same user may need to view the schema differently at different times. The issues and approaches for schema evolution in object-oriented databases are discussed in [66] and the references therein.

Several efforts have been underway to understand good style for object-oriented programs, to support their evolution and reuse. Rochat [98] discusses the importance of good programming style in Smalltalk and proposes several semantic and syntactic guidelines. Lieberherr and others in the Demeter project [71] have proposed an approach for hiding the structure of a class in order to increase maintainability. Johnson and Foote [59] propose design rules to support reusable classes. The authors argue that restructuring an object-oriented program in line with these guidelines will improve its reusability and maintainability.

Very recent research has investigated some of the issues involved in restructuring object oriented programs [15, 31]. Those efforts have tended to focus on inheritance, to achieve such goals as eliminating duplicate definitions of a variable in a program. Refactoring is much more complex that this, however. Refactoring not only involves manipulating inheritance hierarchies but also involves such tasks as renaming a class, splitting up classes and functions, and removing conditional statements. Also, some refactorings involve not only inheritance hierarchies but also instance hierarchies, where an instance of one class is a component of another class [59].

The following chapter introduces the set of refactorings, providing examples of the three high-level refactorings.

Chapter 3

Refactoring Application Frameworks

3.1 Object-Oriented Application Frameworks

An important object-oriented technique to facilitate design-level reuse is an *application framework,* which is an abstract design of an application, consisting of an abstract class for each major component. Frameworks are an emerging technique; there is still much to be learned concerning the design, configuration, and architecture-level description of frameworks. Among the earliest examples of object-oriented application frameworks were the Lisa Toolkit [57] and Smalltalk Model/View/Controller [48]. More recently, several projects have been undertaken at the University of Illinois at Urbana-Champaign to design frameworks; these include the *Typed Smalltalk* (TS) optimizing compiler framework for code generation and optimization [60], the FOIBLE framework for visual programming [58] and the *Choices* object-oriented operating system. *Choices,* written in C++, is more than just an operating system; it is really an operating system framework consisting of interlocking frameworks for file systems [77], virtual memory [103], communication [129], and process scheduling [102].

Good frameworks are usually the result of many design iterations and a lot of hard work involving structural changes [59, 85]. These changes may involve a single refactoring, or a series of related refactorings. The following sections describe examples of several common refactorings. They are mostly based on the analysis of changes made during the iterative design of the *Choices* file system framework. That analysis was based on a series of twelve snapshots of the framework covering a two-year period.

To motivate the practical importance of refactoring in designing application frameworks, the examples presented in the chapter are *real*; that is, the examples describe refactorings that were actually applied (manually) to improve application frameworks in the *Choices* operating system and *Typed Smalltalk* projects. This has the disadvantage that the examples include some extraneous detail and complexity, but has the advantage that the examples show how refactorings are applied in actual design tasks.[1]

The relationships among classes become more explicit and their interfaces become more abstract as a framework matures. A shallow (almost flat) hierarchy of classes evolves into a deeper

[1]Chapter nine describes two simpler, albeit less practical, examples that show how refactorings defined in earlier chapters can be used together.

class hierarchy. Instance hierarchies emerge, where an instance of one class is a component of another class [59]. Often, as instance hierarchies are defined the inheritance hierarchies also change and become easier to understand. Refactorings to generalize and specialize classes in an inheritance hierarchy, or to recognize and restructure instance hierarchies, can support this maturation of an application framework.

3.2 Refactoring To Generalize: Creating An Abstract Superclass

As the design of an application framework matures, general concepts are usually derived from specific examples. Often, these examples are implemented in concrete classes that intertwine the case-specific behavior with more general, common abstractions. As common abstractions are determined, it is useful to separate these abstractions from the example-specific behavior. One way to do this is to define an abstract superclass for a set of concrete classes, and migrate the common behavior to that superclass. This refactoring not only clarifies the design of the framework, but better ensures consistency by defining the abstraction in one place. The concrete classes retain the behavior, although it is now inherited rather than being locally defined.

Abstract classes represent the *protocol* of an object, which is a key part of its specification. An object's protocol is the set of messages that it will accept.[2] Standard protocols support interchangeable components and assist programmers in communicating with one another.

Abstract classes never have instances, and often do not have instance variables. They define *behavior* in terms of a few undefined methods implemented in the subclasses. In a mature inheritance hierarchy, the leaf classes are concrete and the classes closer to the root are more abstract. When defining a new class, it is usually desirable to inherit from an abstract class, to inherit the protocol but not be tied to a particular data representation.

Creating an abstract class is important but not easy. Beck observed that, among Smalltalk researchers, useful abstractions are usually created by programmers who are willing to redo their code several times to produce easy-to-understand and easy-to-specialize classes [86]. Creating reusable abstract classes often involves examining an existing implementation to find an abstraction hidden in a concrete class (or classes), and reorganizing the class hierarchy to make that abstraction explicit [59].

Consider the example, shown in Figure 3.1, of an abstract class created for the *Choices* file system framework [76]. One of the central classes in that framework[3] is MemoryObject [104]. A MemoryObject is a sequence of identically sized blocks. It also is responsible for maintaining the number of blocks it contains. The key operations provided by MemoryObjects are *read*, *write*, and *size*. Many parts of the file system are MemoryObjects, such as files and disks. An Inode[4] contains a description of the disk layout of a file and other information such as the file owner, access permissions and access times [6]. In the *Choices* file system framework, Inode was represented as a subclass of MemoryObject.

[2] The specification of an object also includes class invariants and the relationships with other objects.

[3] The file system has a layered structure that is much more comprehensive than what is presented here; see [76].

[4] Inode is standard UNIX® operating system terminology; it is a contraction of the term *index node.* UNIX is a registered trademark of UNIX Systems Laboratories, Inc.

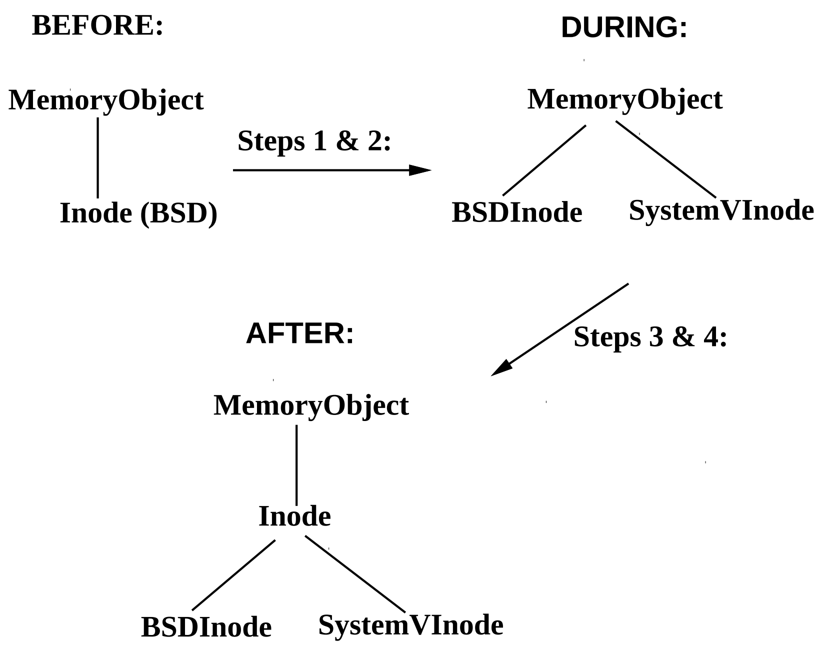


Figure 3.1: Creating An Abstract Superclass

An early version of the *Choices* file system framework only supported the BSD UNIX [37] operating system functions and formats. Thus, instances of the Inode class represented inodes for BSD UNIX file systems. A series of changes, shown in Figure 3.1, were made to generalize the Inode class in order to support both BSD UNIX and UNIX System V [5] file formats:

1. the Inode class was renamed to BSDInode,
2. the SystemVInode class was added as a subclass of MemoryObject; variables and functions were copied from the BSDInode class, and modified,
3. the Inode class was added as a subclass of MemoryObject and as an abstract superclass of BSDInode and SystemVInode,
4. the variables and functions common to BSDInode and SystemVInode were migrated up to their common superclass.

During the fourth step, before some of the common members could be moved, several structural modifications to the subclasses were needed. For example, over 90 percent of the code for the *mapUnit* function in the SystemVInode class was the same as the *mapUnit* function defined in the BSDInode class. However, the code for getting and setting a logical block number was different between the two file system formats, hence there were a few differences in the implementation of the *mapUnit* functions. Before the function could be migrated to the superclass, these differences needed to be taken out of the subclass definitions of the *mapUnit* function. To handle this, the following structural changes were made:

4a) new functions were defined in each subclass to capture the differences

4b) in the definitions of the *mapUnit* function, the differing code was replaced by calls to these functions.

4c) the subclass definitions of the *mapUnit* function now matched, and the function could be migrated to the superclass.

Note that of the four steps listed above for defining the abstract class and adding System V features, only in step two was the functionality of the system changed. The other three steps were refactorings. Step one, where the class was renamed, was clearly needed to distinguish the old file format from the new format, added in step two. Steps three and four added no new features to the system, but were applied to produce an abstraction that allowed additional file system formats to be more easily added later. The file system framework followed an evolution typical of application frameworks: as the framework matured and reusable abstractions were made explicit, new features such as support for the MS-DOS™ file system format and support for persistent objects were more easily integrated into the framework[5].

Note also that in the above example, steps three and four could have come before step two; an abstract superclass could have been defined and some members migrated before the new class was added. The important point here is that restructurings are a critical part of this high-level operation, and that the refactorings can be separated from the operations that add or change functionality in the implementation.

Chapter six defines the high-level operations for defining an abstract superclass.

3.3 Refactoring To Specialize: Subclassing and Simplifying Conditionals

Sometimes a program contains a class that embodies a general abstraction and *several different* concrete cases. In this case, the design of an application framework may be improved by specializing; that is, by defining subclasses corresponding to the cases, and migrating case specific behavior down to the subclasses.

Interestingly, while refactoring to specialize is different in several important ways from refactoring to generalize (discussed in the prior section), the end results of applying these two refactorings are very similar. Generalizing using abstract classes, and specializing using subclassing, both can be used to capture a common abstraction in an abstract superclass and capture the case-specific behavior in the subclasses.

In order to appreciate the need for this refactoring, it is important to understand the role of subtyping for specialization. A *type* characterizes the behavior of its instances by describing the operations that can manipulate those objects [53]. A program with a well-designed type structure will minimize and localize the dependencies among types, enhancing maintainability and extensibility. A *type hierarchy* is composed of subtypes and supertypes. The intuitive notion of a subtype is one whose objects provide all the behavior of another type (the supertype) plus something extra. Subtyping is most frequently used to model conceptual hierarchies. When used in this way, a subtype describes a type that is more specific than a supertype; the subtype is a specialization of the supertype [53, 74].

[5]MS-DOS is a trademark of Microsoft Corporation.

Inheritance can be used in several ways; some clarify the design of a framework while others can make a framework more difficult to understand. One use of inheritance is to model the type hierarchy - that is, represent the supertype as a superclass and a subtype as a subclass. However, inheritance can be used in other ways. Inheritance is sometimes used to support programming-by-difference, where the programmer develops a new class by choosing a similar existing class, inheriting implementation from it and implementing the differences in the new subclass. In this case, the subclass may not be a specialization of the superclass - the resulting class hierarchy may not model any meaningful concept (other than expedient code copying) and can be difficult to understand.

Liskov [74] argues that using inheritance to model the type hierarchy supports data abstraction, which can simplify program maintenance. Liskov notes several uses of supertyping and subtyping in object-oriented systems, including:

- defining a supertype to capture the design insight that a set of existing types are related
- incrementally refining the design of a system by introducing subtype(s) that are refinements of a common abstraction.

Experience with object-oriented application frameworks has suggested that, while subtyping is useful for specializing, the class being specialized often does not start out being abstract, but rather is a concrete class that contains a reusable abstraction. A class sometimes embodies a general abstraction and *several different* concrete cases which are candidates for specialization. Often the behavior that distinguishes the concrete examples is encoded on the state of an object as flags, tags and conditions. As Stroustup notes in [114], representing the behavior in this way may work fine for small programs written by a single person, but has a fundamental weakness: it depends on the programmer manipulating the specializations (types) in a way that cannot be checked by the compiler. Problems arise if the programmer fails to test a type flag, or fails to include all the types in conditional statements. Code that includes the explicit conditional tests can be large and difficult to read.

When the functions defined in a class have conditional statements that each test for the same set of conditions, this may suggest that subclasses should be defined corresponding to those conditions. Or, stated more precisely, when the functions defined in the class have conditional statements that each test for the same set of conditions, a set of subclasses could be defined each of whose corresponding abstraction implies an element in the set of conditions. In each newly defined subclass, the functions can be simplified by replacing the conditional statements with just the code segment for the condition corresponding to (implied by) the subclass.

There are several examples of this type of restructuring that occurred in the *Typed Smalltalk* [60] project and the *Choices* operating system project; it was also useful in converting a program written in the C programming language to C++.

The *Typed Smalltalk* (TS) project is developing an optimizing compiler for a typed version of Smalltalk. In the TS compiler, instances of the FlowNode class, representing a basic block, are containers of a sequence of straight line code (assignments) ending with a statement that alters the flow of control. The ending statement in the sequence could be a return statement, a conditional jump statement or an unconditional jump statement. As the implementation progressed, several functions needed to test the type of the ending statement. For example, one function tested if the ending statement was a conditional jump; if so, and the condition was an invariant for the entire sequence represented by the FlowNode, the conditional jump could be converted to an unconditional jump. Another function tested whether the ending statement was an unconditional jump to a sequence represented by another FlowNode; if so, the two FlowNodes could be merged and replaced by a single FlowNode.

In the early designs, it had not been clear that the type of the ending statement was an important discriminator among FlowNodes. However, as the design matured, the implementation of FlowNode was manually refactored: FlowNode became an abstract superclass and the concrete subclasses UncondJumpFlowNode, CondJumpFlowNode and ReturnFlowNode were added. The conditional statements were simplified. This refactoring made the design easier to read and modify.

A similar refactoring was applied at several stages in the evolution of the *Choices* file system. In an early version, the MSDOSEntry class defined directory entries in the MSDOS format. The *mapUnit* member function included a conditional to handle the case when the file was the root directory. A manual refactoring of the code included adding a new subclass called MSDOSRootEntry and streamlining the *mapUnit* function in the MSDOSEntry and MSDOSRootEntry classes. Later, in the PersistentMemoryObject class there was a conditional that tested whether or not a container attribute was null. To discriminate between the cases, a new subclass FileObject was added. These changes made the design of the Choices File System easier to understand.

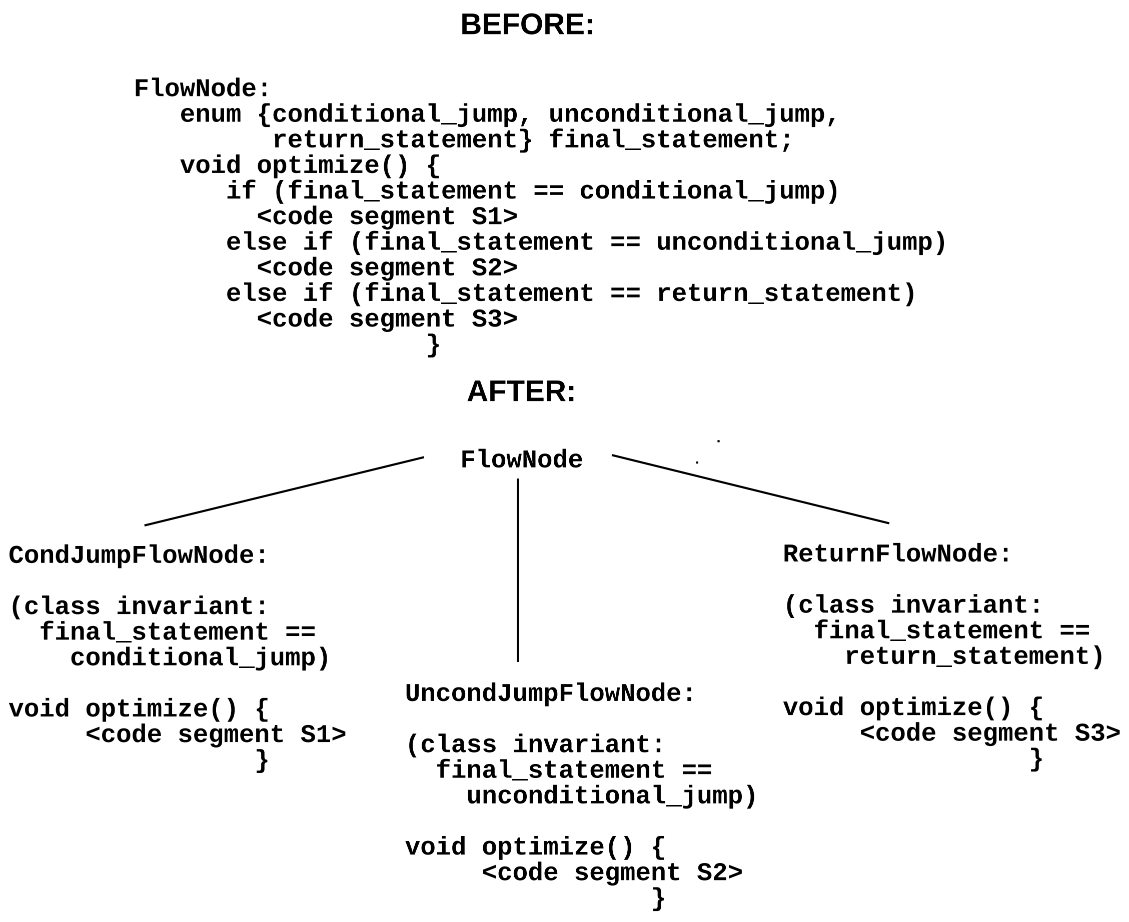
Another benefit that is sometimes realized by replacing conditional tests with subclassing is that the run time performance may improve, as reported by Russo and Kaplan [105]. They ported a Scheme interpreter from the C-language to C++. In a traditional C implementation, the cornerstones are a discrimination union and a procedure body that is essentially a giant switch statement. In their C++ implementation, they distributed the switch statement over classes representing the cases, and found that this made the code more modular and reduced maintenance effort, while *speeding up* the interpreter.

Thus, subclassing, and reducing the conditional statements, may improve both the clarity of the design and the run-time performance. While this refactoring is useful, there are complexities with it, as illustrated in the Figure 3.2, based on the above discussion of FlowNodes in the TS compiler.

In the example, subclasses are added corresponding to the conditions. Put another way, the FlowNode class covered a large number of states, and the newly defined subclasses partition that state space. While in this example it may be intuitively obvious to the programmer how the optimize function should be simplified in each subclass, it would not be obvious to a compiler or a restructuring tool. In *C++* there is no easy way to describe the condition (i.e. set of states) covered by a class. For refactoring, this design information is represented as a *class invariant*, which is a predicate known to be true throughout the lifetime of each instance of a class. In refactoring, a class invariant is assigned to each subclass, and the class invariant is used to simplify the conditional statement. In general, the problems of determining if a class invariant holds for a class, and simplifying a conditional statement given a class invariant, are undecidable. Fortunately, these problems are decidable for many common cases. Chapter seven defines this refactoring.

3.4 Capturing Aggregations and Components

The two previous sections discussed refactorings that refine an inheritance hierarchy. Inheritance is a powerful technique, but in modeling the relationships among classes it is sometimes



BEFORE:

```
FlowNode:
   enum {conditional_jump, unconditional_jump,
         return_statement} final_statement;
   void optimize() {
      if (final_statement == conditional_jump)
        <code segment S1>
      else if (final_statement == unconditional_jump)
        <code segment S2>
      else if (final_statement == return_statement)
        <code segment S3>
                   }
```

AFTER:

```
FlowNode
```

```
CondJumpFlowNode:

(class invariant:
  final_statement ==
    conditional_jump)

void optimize() {
     <code segment S1>
                }
```

```
UncondJumpFlowNode:

(class invariant:
  final_statement ==
    unconditional_jump)

void optimize() {
     <code segment S2>
                }
```

```
ReturnFlowNode:

(class invariant:
  final_statement ==
    return_statement)

void optimize() {
     <code segment S3>
                }
```

Figure 3.2: Simplifying Member Function *optimize*

overused and incorrectly used. This section considers refactorings to support *aggregations*, which model some whole-part associations between objects.

An aggregate object, sometimes called a composite object or a container, is made up of components. Components are stored as member variables in the aggregate object; however, as described below, not all member variables store component objects. One important quality of components needed in refactoring is that a component object cannot be assigned to more than one aggregate object at a time.

Whole-part relationships among objects are sometimes not obvious until implementation is underway. A relationship might first be modeled using inheritance and later is refined into an aggregation. Refactorings can help make aggregations more explicit, and make component classes more reusable.

The following sections describe modeling whole-part relationships, moving members between aggregate and component classes, and converting a relationship modeled using inheritance into an aggregation.

3.4.1 Background: Modeling Whole-Part Relationships

As Wirfs-Brock, Wilkerson and Wiener note [126], inheritance is a natural way to represent some but not all interclass relationships. They discuss three important relationships to examine when structuring an object-oriented system:

1. *is-kind-of*
2. *is-analogous-to*
3. *is-part-of*

If objects of class A are a kind of B, this suggests representing A as a subclass of B. If objects of class A are analogous to objects of class B, this may suggest defining a common abstract superclass that captures their commonalities. However, if an object of class A is a part of an object of class B, inheritance is not a natural way to represent this.

Suppose, for example, a framework is being designed to simulate automobile design and manufacture. The objects *car, door, roof* and *tire* are recognized to be important in this application and classes are defined for them. These objects may share common characteristics such as color and materials used in their manufacture, but it is unnatural to think of a tire or a door as a specialization of an automobile - rather, they are parts of an automobile. A mechanism is needed for representing this whole/part relationship.

The whole-part relationship is a special association between objects, often called an *aggregation.* There are many examples [66, 126] of aggregations:

- a car and its automotive parts
- a company and its departments
- a drawing and its drawing elements
- a program and its program fragments
- a desk lamp and the parts used in its manufacture.

Several tests may indicate whether an association between objects is an aggregation [101]:

- Is one object part of the other?
- Are some operations on the "whole" applied to its "parts"?
- Are some attribute values propagated from the whole to its parts?
- Is there an intrinsic asymmetry to the association, where one object class is subordinate to the other?

The aggregate object, sometimes called a composite object or a container, is made up of components. The aggregate object can be treated as a unit for many operations. One asymmetric aspect of an aggregation is that the aggregate object usually needs to know about its components, but less often does a component need to know either about the aggregate object that contains it, or about other components. During design, recognizing that an association between objects is an aggregation can help determine where the behavior, and the responsibilities for maintaining information, ought to be placed.

The decision to model an association as an aggregation is a matter of judgement. As is typical in modeling, there are few hard and fast rules [101].

While it is common to represent the components of an aggregate object as member variables, not all member variables necessarily represent components. For example, member variables in an Automobile class might represent its color, age or the number of passengers it would accommodate. These member variables represent attributes of an automobile but are not components of it, in the sense that a tire is a component of an automobile. Other associations between objects that do not necessarily imply an aggregation are the depends-upon and needs-to-know-about relationships. For example, in a project management application, an object representing a project task may hold pointers to other task objects that depend upon its completion; when its task completes it send messages to activate those objects. Other objects handling the user interface may hold pointers to the task objects, so that they can query the task objects when a status graph is requested by the user. In each of these cases, the association between objects is different from the whole/part relationship.

There are different kinds of components. A key distinction is whether a component object must be exclusive to one aggregate object, or whether it can be shared by several aggregate objects. For example, in a project management application, tasks and employees are both components of a project. While a task would probably be exclusive to one project, an employee may be shared among several projects. Refactorings involving aggregate objects and their components require that a component object be exclusive to one aggregate object, for reasons described below.

3.4.2 Moving Members Between Aggregate and Component Classes

When an aggregation is recognized, it can help in partitioning behavior between classes. If the classes are already implemented before the aggregation is recognized, the design of the system may be improved by moving members between an aggregate object and its component(s). For example, warranty information stored in the tire component may more appropriately belong in its aggregate object (a car) if the responsibility for managing vehicle maintenance is assigned to the car class. If the tires are components of several different (otherwise unrelated) classes[6], moving behavior out of the tire class may involve adding it to *several* aggregate classes, which can be complicated.

For example, moving a variable from an aggregate class to the class of one of its components is behavior preserving only where is a one-to-one relationship between each instance of the aggregate class and instances of the class to which the variable is being moved. Otherwise, if a component object were shared among more than one aggregate object then the single new variable in the (shared) component would be replacing a set of variables, one in each of the aggregate objects that contained the component; information could be lost and hence the refactoring would not be behavior preserving.

In order for refactorings involving aggregates and their components to be behavior preserving, each component must be exclusive to one aggregate object. The reasons for this are detailed in

[6]For example, tires are used both in automobiles and in automatic baseball pitching machines; these two uses are otherwise unrelated.

chapter eight. Before a member variable can be designated as a component, it must be shown that any instances assigned to it are not simultaneously assigned to another variable designated as a component. A method for checking these constraints is described in chapter eight.

3.4.3 Converting Associations Modeled Using Inheritance Into Aggregations

Sometimes an aggregation is not obvious until after the implementation is underway. The "whole" may bear some behavioral similarities to its parts, and those similarities may initially be modeling using inheritance. A useful refactoring to support framework evolution is converting a superclass/subclass relationship into an aggregation, as shown in Figure 3.3.

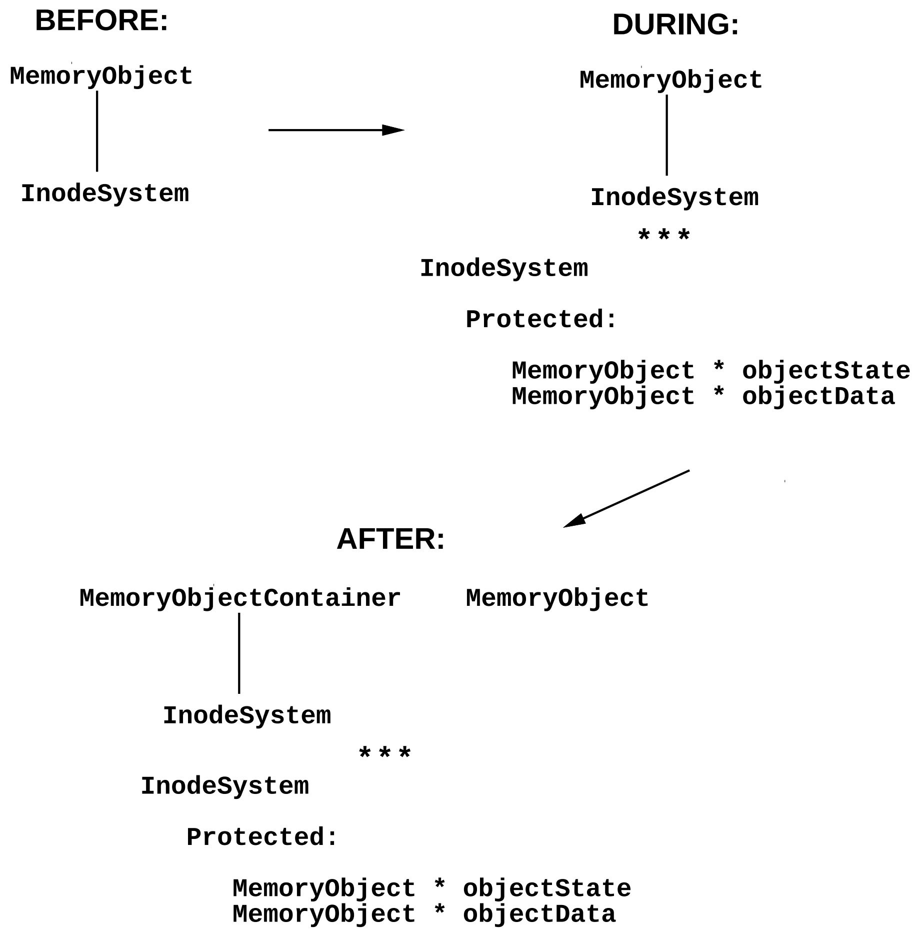


Figure 3.3: Superclass Converted to Component

In the *Choices* file system, an InodeSystem managed the creation of and access to Memory-Objects. It managed several high-level operations regarding a file system; these operations required low level support (*i.e.* read and write operations) for accessing and storing pointers to its MemoryObjects. In an early version of the file system framework, InodeSystem was defined as a subclass of MemoryObject, to inherit low level operations such as read and write. However, InodeSystem really represented a different abstraction from a MemoryObject. As the design of the framework was refined, component member variables, which were instances of the MemoryObject class, were added to InodeSystem to provide the low level support for its file system operations. The high level operations defined in InodeSystem were changed to invoke behavior in the new components. It was now no longer necessary for InodeSystem to inherit the low level operations from MemoryObject. The superclass of InodeSystem was changed. The notion of a MemoryObjectContainer was introduced. MemoryObjectContainers kept track of partitioning a large MemoryObject into a set of smaller MemoryObjects (i.e. a disk into a set of files). InodeSystem was really a specialization of MemoryObjectContainer, and became one of its subclasses.

Both before and after the refactorings were applied, the MemoryObject class provided operations to the InodeSystem class. Before the refactoring, the services were inherited by the InodeSystem class; later, they were provided by its components.

From a software engineering viewpoint, there are advantages in using aggregations and components, as opposed to inheritance, to accomplish reuse. Data abstraction supports reuse [74]. An aggregate object only sees the public interface of its components, whereas a subclass sees the internals (i.e. a less abstract view) of its superclass. To use the terminology from the software testing literature,[7] inheritance provides a white box (or possibly a grey box) interface, whereas the interface provided by a component to its aggregate object is more typically a black box interface. Black box interfaces provide better encapsulation and abstraction than do white box or grey box interfaces [59].

Thus, while inheritance plays an important role in an application framework, so too can aggregations and components.

There are some complexities in converting a subclass/superclass relationship to an aggregation. Checks are needed to ensure that type requirements on variable assignments and function calls are still satisfied. Previously inherited behavior may need to be copied into one of the classes. Chapter eight describes several high-level operations involving aggregations.

3.5 Supporting Refactorings

The prior three sections focused on the three highest level refactorings: creating an abstract superclass, subclassing and simplifying conditions, and creating aggregations and reusable components. As noted in chapter one, the high-level refactorings are supported by a set of twenty six low-level refactorings, which are listed below.

All but the final category of the supporting refactorings listed below are *atomic*; that is, they are the most primitive refactorings. The atomic refactorings create, delete change and move entities. The final category of less primitive (composite) refactorings support slightly more powerful refactoring operations, such as abstracting access to a member variable.

1. Creating a Program Entity:
   (a) creating an empty class
   (b) creating a member variable
   (c) creating a member function.

[7]In the software testing literature, black box denotes functional testing and white box denotes structural testing. The two types of testing are compared in [13].

2. Deleting a Program Entity:
    (a) deleting an unreferenced class
    (b) deleting an unreferenced variable
    (c) deleting a set of member functions.
3. Changing a Program Entity:
    (a) changing a class name
    (b) changing a variable name
    (c) changing a member function name
    (d) changing the type of a set of variables and functions
    (e) changing access control mode
    (f) adding a function argument
    (g) deleting a function argument
    (h) reordering function arguments
    (i) adding a function body
    (j) deleting a function body
    (k) convert an instance variable to a variable that points to an instance
    (l) convert variable references to function calls
    (m) replacing statement list with function call
    (n) inlining (ie inline expanding) a function call
    (o) changing the superclass of a class
4. Moving a Member Variable:
    (a) moving a member variable to a superclass
    (b) moving a member variable to a subclass.
5. Less primitive (composite) refactorings:
    (a) abstract access to a member variable
    (b) convert a code segment to a function
    (c) moving a class.

These low-level refactorings are described in chapter five.

This list, combined with the three high-level refactorings, is an elaboration of the original list of eight refactorings described in section 1.2.1 (and in [85]). From that original list:

- the first three refactorings map to the high-level refactorings, which are implemented using many of the low-level refactorings listed above
- the fourth refactoring (moving a class) maps to refactoring 5(e)

- the fifth refactoring (moving variables and functions) maps to refactorings 1(c), 2(c), 4(a), and 4(b)
- the sixth refactoring (replacing code segment with function call) maps to refactoring 3(m)
- the seventh refactoring (changing names) maps to refactorings 3(a), 3(b) and 3(c)
- the eighth refactoring (abstracting variable access) maps to 5(a).

It is interesting to note that, while the list of refactorings was determined mostly from analyzing C++ programs, many should be applicable for refactoring programs written in other object-oriented languages. Only a small subset appear to be C++-specific.[8] Applying these refactorings to other object oriented languages is an area for future research.

3.6 Language Features in Refactoring

Below is a brief summary of the language features upon which the refactoring descriptions are based. Chapter four discusses the related issues of the domains of refactoring and behavior preservation.

A program consists of a single global function called *main*, and (possibly null) sets of classes, global enumerated types and global variables.

The scope of each class is the entire program. A class has at most one direct superclass. A class consists of a set of member variables and set of member functions. All member variables have distinct names, as do all member functions. All variables and functions are typed. The type of a member variable can be a class, in which case an instance of that class is embedded in the variable's containing class when that class is compiled. The type of a member variable can be a pointer to a class, in which case it can be assigned an instance of that class or an instance of one of its subclasses. A function defined in a superclass can be redefined in a subclass; all functions that are redefined in subclasses are (using C++ terminology) virtual.

The language features covered are a major subset of C++. However, several features of C++ and other object-oriented languages were not included here. Multiple inheritance and overloaded member function names (where two member functions of a class can have the same name but differ in argument types) were not included. They would have complicated the precondition checking for refactorings, particularly as related to naming conflicts, and made the refactoring descriptions tedious and more difficult to understand. Multiple inheritance is not a feature in Smalltalk, nor was it used in the versions of the *Choices* file system analyzed in this research. Overloaded member function names can be mapped into distinctly named functions by concatenating function names and arguments types, as is done in some C++ compilers. Another feature of C++ not included here is type casts. Explicit type casts is generally recognized as poor coding practice. Handling type casts would require more complex flow analysis than is presented here and is an area for future research. Finally, C++ is an evolving language; parameterized classes, a very recently added feature, is not included in this analysis.

[8] Three refactorings that would not apply for Smalltalk programs are 3(d),(e), and (k). These refactorings change type, access control mode, and convert an instance variable to a variable that points to an instance.

3.7 Summary: Importance & Complexity of Refactoring

Refactoring is important. As noted earlier, in the view of Deutsch [40] and others, the key intellectual content of software is captured in the factoring of functions and in the design of interfaces. Refactoring helps in making this intellectual content more explicit and, hence, helps in designing reusable software. Experienced developers of object-oriented systems find themselves restructuring their implementations, although they may not be consciously aware that refactoring tasks are part of their development process.

Refactoring is complex. Much of the complexity is due to the interrelationships among classes. For example, when a class is moved and a subclass/superclass relationship is changed to a container/component relationship, checks and structural changes are often needed to retain inherited behavior and satisfy type requirements on assignments and function calls. As a system grows, doing these tasks by hand becomes more complicated, tedious and error-prone.

The following chapters describe a way to automate the process of refactoring.

Chapter 4

Preserving Behavior During Refactoring

Intuitively, refactorings should preserve the behavior of a program. This chapter discusses several topics related to behavior preservation. Section 4.1 describes several program properties that are easily violated in refactoring. Section 4.2 defines the domains of the arguments to the refactorings. Section 4.3 lists the functions used to describe preconditions of the refactorings; the preconditions ensure that behavior is preserved by the refactoring.

4.1 Program Properties and Behavior Preservation

After a refactoring, a program must be syntactically correct. For example, the superclass of a new class must be an existing class. Or, a function in a subclass that overrides a function defined in its superclass must be type compatible with the corresponding function in the superclass.

A compiler could catch these errors. Therefore, one way to prevent these errors from happening would be to save the current version of a program before each refactoring, apply the refactoring without regard to these errors, and then recompile the program. If an error is flagged, fall back to the old version.

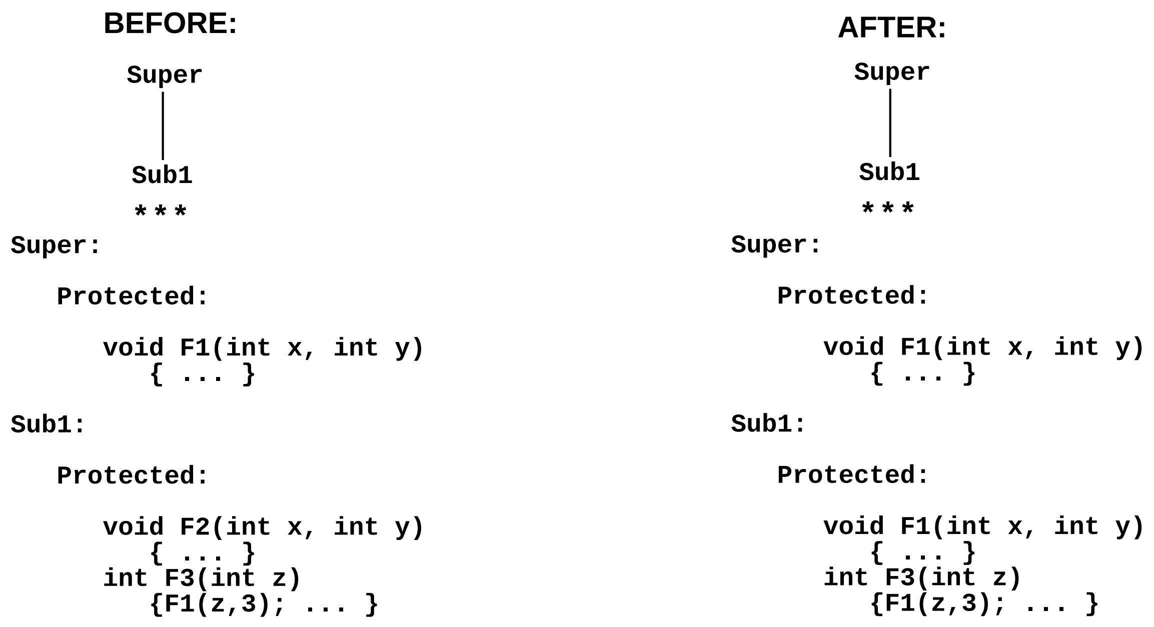
However, there are two major problems with this approach:

1. the approach might be unacceptably slow, especially for higher level refactoring operations that involve a series of more primitive refactorings.

2. more importantly, there are some errors that could change the behavior of the program but would not be picked up by the compiler.

Consider the erroneous refactoring shown in Figure 4.1. Function *F1* is a protected member in class *Super*, while *F2* is defined in class *Sub1*, one of its subclasses. The argument and return types of *F2* are (coincidently) the same as those of *F1*. In class *Sub1*, function *F3* includes a call to *F1*, which is inherited from the superclass.

A refactoring is applied, in class *Sub1*, to rename function *F2* to *F1*. This renaming would result in a syntactically correct program (it would compile cleanly). However, the behavior of the program has probably changed. Function *F3* would now call the newly renamed function defined in its local class, rather than the function previously inherited from the superclass.



```
***
Super:

   Protected:

      void F1(int x, int y)
         { ... }

Sub1:

   Protected:

      void F2(int x, int y)
         { ... }
      int F3(int z)
         {F1(z,3); ... }
```

```
***
Super:

   Protected:

      void F1(int x, int y)
         { ... }

Sub1:

   Protected:

      void F1(int x, int y)
         { ... }
      int F3(int z)
         {F1(z,3); ... }
```

Figure 4.1: Erroneous Renaming Of Member Function F2

Assuming that the two functions behave differently, the behavior of function *F3* program would have changed. A compiler would not pick up this error.

Without rules for checking that these program properties are not violated, it is possible to produce a program that either is not syntactically correct or (worse) compiles correctly but behaves differently after the refactoring.

During the research prototyping, a particular set of syntactic and semantic properties of programs (listed below) was found to be easily violated if explicit checks were not made before a program was refactored. In the refactoring definitions, behavior preservation is argued in terms of these program properties.[1] These properties relate to inheritance, scoping, type compatibility and semantic equivalence. Recompiling the program after refactoring could flag violations of the first six properties, but violations of the seventh property might not be flagged. The properties are:

1. *Unique Superclass.* After refactoring, a class must always have at most one direct superclass and its superclass must not also be one of its subclasses. The focus of this research was upon single inheritance systems, without cycles in the inheritance graph.[2]

2. *Distinct Class Names.* After refactoring, each class must have a unique name. In this research, classes are not nested; that is, the scope of each class is the entire program.

[1]They are similar to the properties analyzed in [9, 31]. Those efforts focused on restructurings involving program data, while this research considers structural changes to both functions and data. The C++ language is a semantically complicated language, supporting machine level operations such as pointer arithmetic; these complexities make it difficult to more precisely define what behavior preservation means for C++ programs.

[2]Smalltalk is a language with these features, and the C++ based application framework analyzed in this research had these features.

3. *Distinct Member Names.* After refactoring, all member variables and functions *within a class* have distinct names. However, this does allow, for example, a member function in a superclass to be overridden in a subclass.

4. *Inherited Member Variables Not Redefined.* A member variable inherited from a superclass is not redefined in any of its subclasses.

5. *Compatible Signatures in Member Function Redefinition.* After refactoring, if a member function defined in a superclass is redefined in a subclass, all attributes (except the function body) of the two functions must be compatible. All functions are (using C++ terminology) virtual, and can be overridden in subclasses. Signatures of virtual functions in C++ are compatible only if their (return and argument) types match exactly [41].[3]

   This allows a function defined in a superclass to be overridden in a subclass, as long as the signatures are compatible. The ability to override inherited functions is important in refactoring. For example, in defining an abstract class it is sometimes useful to define a function in the superclass that does not contain a function body; the function is overridden in subclasses, where the function body is defined.

6. *Type-Safe Assignments.* After a refactoring, the type of each expression assigned to a variable must be an instance of the variable's defined type, or (if it is a pointer variable) possibly an instance of one of its subtypes. (In C++, subtyping is implemented using subclassing.) ' This applies both to assignment statements and function calls.

7. *Semantically Equivalent References and Operations.* This topic is discussed below.

4.1.1 Semantically Equivalent References and Operations.

As noted above, saying that refactorings are behavior preserving is more than saying that they produce legal programs. The versions of the program before and after a refactoring must also produce *semantically equivalent references and operations.* Semantic equivalence is defined here as follows: let the external interface to the program be via the function *main.* If the function *main* is called twice (once before and once after a refactoring) with the same set of inputs, the resulting set of output values must be the same.

This definition of semantic equivalence allows changes throughout the program, as long as this mapping of input to output values remains the same. Imagine that a circle is drawn around the parts of a program affected by a refactoring. The behavior as viewed from outside the circle does not change. For some refactorings, the circle surrounds most or all of the program. For example, if a variable is referenced throughout a program, the refactoring that changes its name will affect much of the program. For other refactorings, the circle covers a much smaller area; for example, only part of one function body is effected when a particular function call contained in it is inline expanded. In both cases, the key idea is that the results (including side effects) of operations invoked and references made from outside the circle do not change, as viewed from outside the circle.

This allows for several important changes that don't affect equivalence:

[3]Signature compatibility has different meanings in other object oriented languages such as Eiffel® [79]. Eiffel is a registered trademark of Interactive Software Engineering, Inc.

- expressions can be simplified and dead code removed. Conditional statements can be simplified based on invariant conditions known when the conditional is encountered (eg. class invariant). Variables, functions and classes can be removed if they are unreferenced.
- similarly, variables, functions and classes can be added if they are unreferenced.
- the type of a variable can be changed by a refactoring, as long as each operation referenced on the variable is defined equivalently for its new type.
- references to a variable and function defined in one class can be replaced by references to an equivalent variable or function defined in another class. One implication of this is that locally defined members can be replaced by inherited members (and vice-versa) provided that the member declarations are equivalent. In addition, references to members of an aggregate object can be replaced by references to members of one of its exclusive component (and vice-versa).

One very restrictive way to ensure that the program remains semantically equivalent across a refactoring is to require that after a refactoring all references are to the same variables and functions defined in the same classes as before the refactoring. There are a few refactorings that this would permit, such as adding a new variable or renaming an existing one. However, it would prevent many useful refactoring operations.

Behavior can be preserved under less restrictive conditions, when for example a class member is moved up or down an inheritance hierarchy, or when a member is moved between an aggregate class and a component class. When a member variable is moved from one class to another, behavior will be preserved if there is a strict one-to-one relationship between the old variable and new variable:

- in the new class the variable has the same type and lifetime as before,
- all references to the old variable are changed to refer to the new variable,
- the new variable is not otherwise referenced.

These preconditions are more easily checked for refactorings involving inheritance than for refactorings involving aggregates and components, as is detailed in the later chapters.

Some refactorings change the size of objects or change the relative physical positions of variables within an object. These refactorings are behavior preserving only for well behaved C++ programs. Consider the physical layout of a class in C++, in which the variables inherited from a superclass precede variables locally defined. The refactoring that moves a variable from a subclass to superclass is behavior preserving as long as the program is not dependent upon the physical ordering of variables. If a program then performs integer arithmetic with the address of a member variable, then the behavior of the program may change when the variable is moved. Similarly, consider the refactoring that moves a variable from an aggregate class to a component class. In this case, the size of each class changes. If a program tests the physical size of an object, the behavior of the program may change when the variable is moved.

The refactoring approach defined in this thesis does not apply to programs that are dependent upon the physical layout and size of objects. Fortunately, most object oriented programs are not dependent upon these features. A major attractiveness of object-oriented languages is that they provide a level of abstraction that shields the user from the underlying representation of objects; it is generally considered bad style to write programs that depend on this underlying representation. Refactorings that can change the physical size and layout of objects are noted as such in later chapters of the thesis.

4.1.2 Summary: Program Properties and Behavior Preservation

In summary, refactorings preserve the behavior of a program. This implies that refactorings always result in legal programs that perform operations equivalent to before the refactoring. In this thesis, behavior preservation is argued in terms of the seven program properties described above. These properties are the ones found to be most frequently violated when refactoring programs.

Chapter five describes the low-level refactorings. There, behavior preservation is argued in terms of the program properties described above. Chapters six through eight describe the high-level refactorings. Behavior preservation of those refactorings is mainly argued in terms of the lower level refactorings used to compose it.

4.2 Domains of Refactorings

The arguments to refactorings are typed; the types and their attributes are defined below.

Associated with each variable and function is an attribute that indicates its scope. There are four kinds of variables in C++: global variables, class member variables, argument variables in a function, and variables local to a statement list of a function. Two kinds of C++ functions are considered in this research: a single global function *main()* and class member functions.

When adding a new variable, a name collision occurs if, within the scope of the new variable, there is a reference to a variable with the same name defined in an enclosing scope. For the purposes of checking name collisions, the scope of a variable is as follows. The scope of a global variable is the entire program *Program*. The scope of a class member variable is the class that contains it, if the variable's accessControlMode is private; otherwise, the containing class and its subclasses.[4] The scope of an argument variable is the function that contains it. Similarly, the scope of a local variable is its containing block.

Functions are restricted to a single global function *main()* and a set a member functions for each class.[5] Functions defined in a superclass can be overridden in a subclass, as long as either the superclass function is not referenced within the scope of the class that will override it, or the new function is equivalent with the function it replaces.

Attributes are listed with their allowable type(s) followed by their name:

- program:
  - (set of class) classes
  - function globalFunctionMain[6]

[4]In cases where the variable's accessControlMode is public, the variable can be referenced elsewhere in the program, but, outside the variable's containing class and its subclasses, references to the variable will be prefixed with the name of the object containing the member variable.

[5]This restriction to a *single* global function was made to avoid complicating the notation of some refactorings. A straightforward extension could allow multiple global functions.

[6]C++ allows multiple global functions. Here, a program consists of a single global function *main*, which is not renamable. This restriction, which is not significant, was made to avoid complicating the notation of some refactorings. A straightforward extension could allow multiple global functions.

  - (set of variable) globalVariables of their arguments; the functions are argument domains are defined here.
  - (set of enumeratedType) globalEnumeratedTypes

- class:
  - class superclass
  - string name
  - (set of function) locallyDefinedMemberFunctions
  - (set of variable) locallyDefinedMemberVariables
  - (set of variable) SetOfComponents
  - predicate classInvariant

- enumeratedType:
  - string name
  - (set of string) enumeratedValues

- function:
  - (class + program) owner
  - string name
  - type returnType
  - (list of variable) arguments
  - statement body.
  - (nil + private + protected + public) accessControlMode

- variable:
  - (statement + function + class + program) owner
  - string name
  - type type
  - (set of constant) initializationArguments
  - (nil + private + protected + public) accessControlMode

- type: program.classes + PrimitiveTypes.
  - string name

- statement:
  - (statement + function) owner
  - (list of (statement + expression)) components

- expression: assignment + variableRef + functionCall + dynamicObjectExpression

- assignment:

  - (expression + statement) owner
  - variable destination
  - expression source
  - type type

- dynamicObjectExpression:
  - class instantiatedClass
  - (list of expression) parameters

- functionCall:
  - (expression + statement) owner
  - function calledFunction
  - (list of expression) parameters
  - type type
  - expression prefix.[7]

- variableRef:
  - (expression + statement) owner
  - variable refdVariable
  - expression prefix.[8]

- predicate: used in defining class invariants; defined in chapter six.

In the refactoring definitions, domain attributes are referenced using a *dot* notation. For example, the return type of member function F would be referenced by $F.returnType$.

4.3 Functions For Describing Preconditions

Most refactorings are only behavior preserving under certain preconditions. For example, the type of a variable can be changed only if assignments involving the variable would remain type safe.

Refactorings have their preconditions described in terms of a set of functions. The functions is defined below. There is a small set of primitive functions, followed by a larger set of non-primitive functions. For the non-boolean functions, the return type is listed before the name.

In the preconditions of each refactoring, the function names and arguments are usually descriptive. The definitions in this section, included for completeness, provide a level of detail beyond what may be of interest to some readers. This section can be skipped without missing the intent of the refactorings.

[7] If calledFunction is called through the public interface of an object, this attribute contains the expression that identifies that object; otherwise, this attribute is nil.

[8] If refdVariable is a member variable referenced through the public interface of an object, this attribute contains the expression that identifies that object; otherwise, this attribute is nil.

4.3.1 Boolean Primitive Functions

1. compilesP (function F, (class + program) Scope) $\equiv$
   F compiles within its Scope.[9]
2. convertibleToFunctionP (statement S) $\equiv$
   S can be converted to a legal function (in containingClass(S)).[10]
3. qualifiesAsComponentP (variable V) $\equiv$
   V qualifies as a component member variable of its containing class.[11]
4. noSideEffectsP (expression E) $\equiv$
   E is known to have no side effects.
5. satisfiesClassInvariantP (expression E, class C) $\equiv$
   instances created by E satisfy the class invariant for C.[12]
6. semanticallyEquivalentP ((function or statement) F1, (function or statement) F2) $\equiv$
   F1 is semantically equivalent to F2.
7. subtypeP(type T1, type T2) $\equiv$
   T1 is a subtype of T2.
8. unrefdOnInstancesP(function F, class C) $\equiv$
   function F is not referenced on instances of class C.

4.3.2 Other Primitive Functions

1. firstConditionImpliedByInvariant (statement S (a conditional), predicate P)
   returns the first condition in S implied by P.[13]

4.3.3 Non-primitive Boolean Functions

1. inheritedMemberFunctionNamedP(class C, string S) $\equiv$
   $\exists$ member $\in$ inheritedMembers(C) $\wedge$
      member.name = S.
2. matchingAttributesP(variable V1, variable V2) $\equiv$
   (V1.name = V2.name) $\wedge$
   (V1.type = V2.type) $\wedge$
   (V1.initializationArguments = V2.initializationArguments) $\wedge$
   (V1.accessControlMode = V2.accessControlMode)

[9]This is checked when adding a new function, or adding a function body to a function definition.

[10]S must not define local variables that are referenced outside S. There must be no branches from S to other parts of its current function; nor may there be any branches into the middle of S (if S is a compound statement). Functions and variables visible to S will be visible to the new function, since functions and non-local variables visible to S would be visible to the new function since it is defined in the same class; referenced variables local to the function but not local to S would be passed as arguments.

[11]This would be determined using the algorithms defined in section 8.2.

[12]This would be determined using the algorithm defined in section 7.8.

[13]This would be determined using the algorithm defined in section 7.6.

3. matchingAttributesP(function F1, function F2) $\equiv$
   matchingSignatureP(F1, F2) $\wedge$
   (F1.body = F2.body).

4. matchingSignatureP(function F1, function F2) $\equiv$
   (F1.name = F2.name) $\wedge$
   (F1.returnType = F2.returnType) $\wedge$
   (F1.accessControlMode = F2.accessControlMode) $\wedge$
   length of F1.arguments = length of F2.arguments $\wedge$
   for i = 1 to (length of F1.arguments),
   F1.arguments[i].type = F2.arguments[i].type.

5. memberFunctionNamedP(class C, string S) $\equiv$
   $\exists$ F $\in$
   (C.locallyDefinedMemberFunctions $\cup$ inheritedMemberFunctions(C)) $\wedge$
   F.name = N.

6. redundantIfAddedP(function F, class C) $\equiv$
   $\exists$ F2 $\in$ (C.superclass).locallyDefinedMemberFunctions $\cup$
   inheritedMemberFunctions(C.superclass)) $\wedge$
   F.name = F2.name $\wedge$
   matchingAttributesP(F, F2).

7. varNameCollisionP(string S, (class + function + statement) Scope) $\equiv$
   $\exists$ collidingVar $\in$ Program $\wedge$
   collidingVar.name = S $\wedge$
   Scope $\subset$ scopeOf(collidingVar).

4.3.4 Other Non-primitive functions

1. (set of function) allFunctions(program P) $\equiv$

$$P.globalFunctionMain \bigcup (\cup_{class \in P.classes} class.locallyDefinedMemberFunctions)$$

2. (set of function) allVariables(program P) $\equiv$
   returns the set of all variables declared in P.

3. (set of functionCall) callsTo(function F1) $\equiv$

$$\bigcup_{F2 \in allFunctions(Program)} \{FC \mid FC \in F2.statement \wedge FC.calledFunction = F1\}$$

4. (set of class) classesInternallyReferencing((function + variable) X) $\equiv$

$$\{containingClass(ref) \mid (ref \in referencesTo(X)) \wedge (ref.prefix = nil)\}$$

   (ie the set of classes internally referencing X.)

5. (set of class) classesPubliclyReferencing((function + variable) X) ≡

$$\{containingClass(ref) \mid (ref \in referencesTo(X)) \wedge (ref.prefix \neq nil)\}$$

(ie the set of classes referencing X through the public interface of its containing class(es).)

6. (set of class) classesReferencing(class X) ≡

$$\{containingClass(ref) \mid ref \in referencesTo(X)\}$$

(ie the set of classes containing references to class X.)

7. (set of class) classesReferencing((function + variable) X) ≡
    classesInternallyReferencing(X) ∪ classesPublicallyReferencing(X).
   (ie the set of classes containing references to X.)

8. function collidingFunction (class C, function F) ≡
    returns the function whose name would collide with F if it were added to C.

9.
```
(nil + class) containingClass(
    (variable + function + (set of statement) + expression) containedItem) ≡
        if ((containedItem.owner ∈ Program.classes) ∨
            (containedItem.owner = nil)), /* global function */
                containedItem.owner,
        else containedClass(containedItem.owner).
```

10.
```
(nil + function) containingFunction(
    ((set of statement) + expression) containedItem) ≡
        if ((containedItem.owner ∈ allFunctions(Program)),
            containedItem.owner,
        else containingFunction(containedItem.owner).
```

11. (set of class) directSubclassesOf(class C) ≡

$$\{subClass \in P.classes \mid subClass.superclass = C\}$$

12. directSuperclassOf(class C) ≡ C.superclass

13.
```
(set of variable) expressionsAssignedTo(variable V) ≡
    (∀ assignment expression ∈ scopeOf(V),
        where destination = V, return source) ∪
    (∀ function call ∈ scopeOf(V),
        if V is an argument variable of that function,
            return corresponding value passed as a parameter).
```

14.
```
(set of expression) expressionsAssignedToArgument(variable argVar)
    ∀ functionCall ∈ callsTo(argVar.owner),
        return the expression in the position corresponding to the argument.
```

15. (set of function) functionsCalledBy(function F) ≡
    the set of all function calls in F.statement.

16. (set of function) functionsThatOverride (function F) $\equiv$

$$\{F2 \in allFunctions(P) \mid F2.owner \in subclassesOf(F.owner) \wedge F.name = F2.name\}$$

17. (set of (variable + function)) inheritedMembers(class C) $\equiv$
    inheritedMemberFunctions(C) $\cup$ inheritedMemberVariables(C).

18. (set of function) inheritedMemberFunctions(class C) $\equiv$
```
    if C.superclass=nil, return nil
    else return F $\in$
        (inheritedMemberFunctions(C.superclass) $\cup$
        C.superclass.locallyDefinedMemberFunctions),
    where (F.accessControlMode $\neq$ private) $\wedge$ $\sim$localMemberFunctionNamed(F,C).
```

19. (set of variable) inheritedMemberVariables(class C) $\equiv$
```
    if C.superclass=nil, return nil
    else return V $\in$
        (inheritedMemberVariables(C.superclass) $\cup$
        C.superclass.locallyDefinedMemberVariables),
    where (V.accessControlMode $\neq$ private) $\wedge$ $\sim$localMemberVariableNamed(V,C).
```

20. (set of (function + variable)) locallyDefinedMembersOf(class C) $\equiv$
    (C.locallyDefinedMemberFunctions $\cup$
        C.locallyDefinedMemberVariables)

21. (set of variable) localVariablesIn(function F) $\equiv$
    the set of all local variables in F.statement.

22. (function + nil) memberFunctionNamed(class C, string S) $\equiv$
    F, where F $\in$ (inheritedMemberFunctions(C) $\cup$ C.locallyDefinedMemberFunctions)
        $\wedge$ F.name = S.

23. (function + nil) memberVariableNamed(class C, string S) $\equiv$
    V, where V $\in$ (inheritedMemberVariables(C) $\cup$ C.locallyDefinedMemberVariables)
        $\wedge$ V.name = S.

24. (set of(variable + function)) membersOf(class C) $\equiv$
    inheritedMemberFunctions(C) $\cup$ inheritedMemberVariables(C) $\cup$
        C.locallyDefinedMemberFunctions $\cup$ C.locallyDefinedMemberVariables.

25. (set of (expression + variable)) referencesTo(class C) $\equiv$
    instancesOf(C) $\cup$

$$\{V \mid (V \in allVariables(Program)) \wedge (V.class = C)\}$$

26. (set of functionCall) referencesTo(function F) $\equiv$
    callsTo(F)

27. (set of variableRef) referencesTo(variable V) $\equiv$

$$\bigcup_{V2 \in allVariables(Program)} \{VR \mid VR \in F2.statement \wedge VR.refdVariable = V\}$$

28. (statement + function + class + program) scopeOf((variable + function) item1) ≡
    if item1.owner ∉ Program.classes, /* Global, local or argument variable */
        item1.owner,
    else if item1.accessControlMode = private,
        item1.owner,
    else (item1.owner ∪ subclassesOf(item1.owner)).

29. (set of variable) setOfComponentVariables(class C) ≡

$$\bigcup_{class \in Program.classes} \{V \mid V \in class.SetOfComponents \wedge V.type = C\}$$

30. (set of class) subclassesOf(class C) ≡
    the transitive closure of directSubclassesOf(C).

31. (set of class) superclassesOf(class C) ≡
    the transitive closure of directSuperclassOf(C).

32. (set of variable) variablesAssigned(function F) ≡
    (∀ assignment expression ∈ scopeOf(F),
        where source = a call to F, return destination) ∪
    (∀ function call ∈ scopeOf(V),
        if a call to F is passed as a parameter,
            return corresponding function argument variable).

33. (set of variable) variablesAssigned(variable V) ≡
    (∀ assignment expression ∈ scopeOf(V),
        where source = V, return destination) ∪
    (∀ function call ∈ scopeOf(V),
        if V is passed as a parameter,
            return corresponding function argument variable).

34. (set of variable) variablesAssigned(dynamicObjectExpression E) ≡
    (∀ assignment expression ∈ Program,
        where source = E, return destination) ∪
    (∀ function call ∈ Program,
        if E is passed as a parameter,
            return corresponding function argument variable).

35. (set of variable) variablesReferencedBy(function F) ≡

$$\{VR.refdVar \mid (VR \in F)\}$$

4.4 Summary

Intuitively, refactorings should preserve the behavior of a program. This chapter discusses the program properties that were found to be easily violated in refactoring. Also defined here are the domains of the arguments to the refactorings, and the functions used for defining the preconditions of the refactorings.

Chapter 5

Low-Level Refactorings

This chapter defines twenty six low-level refactorings. They support the three high-level refactorings, described in chapters six through eight. These refactorings are designed to be simple, so that in most cases it is trivial to show that they are behavior-preserving. The behavior-preserving nature of the high-level refactorings is argued in terms of these refactorings.

For completeness, the definition of each refactoring details its preconditions and argues that it is behavior preserving. This information, included for completeness at the end of each refactoring, provides a level of detail beyond what may be of interest to some readers. It can be skipped without missing the intent of the refactoring.

The refactorings defined in this chapter are:

1. Creating a Program Entity:
   (a) *create_empty_class*
   (b) *create_member_variable*
   (c) *create_member_function.*
2. Deleting a program entity:
   (a) *delete_unreferenced_class*
   (b) *delete_unreferenced_variable*
   (c) *delete_member_functions*
3. Changing a program entity:
   (a) *change_class_name*
   (b) *change_variable_name*
   (c) *change_member_function_name*
   (d) *change_type*
   (e) *change_access_control_mode*
   (f) *add_function_argument*
   (g) *delete_function_argument*

   (h) *reorder_function_arguments*
   (i) *add_function_body*
   (j) *delete_function_body*
   (k) *convert_instance_variable_to_pointer.*
   (l) *convert_variable_references_to_function_calls*
   (m) *replace_statement_list_with_function_call*
   (n) *inline_function_call*
   (o) *change_superclass*

4. Moving a member variable:
   (a) *move_member_variable_to_superclass*
   (b) *move_member_variable_to_subclasses*

5. Intermediate level (composite) refactorings:
   (a) *abstract_access_to_member_variable*
   (b) *convert_code_segment_to_function*
   (c) *move_class*

5.1 Creating a Program Entity

These refactorings create a new class or a new class member. Each refactoring is behavior preserving because the entity (class or class member) it adds either is unreferenced or replaces an identically defined, inherited member.

A. *create_empty_class*

Define a new class, with no locally defined members. If a superclass is specified as an argument, the new class becomes its direct subclass.

Arguments: string newClassName, (optional) class superclass.

Preconditions:

1. $\forall$ class $\in$ Program.classes,
   class.name $\neq$ newClassName.
   (the name does not clash with an already existing class)

The arguments to the refactoring specify that the new class will have 0 or 1 direct superclass, satisfying the unique superclass property (property one). The precondition ensures distinct class names (property two). Other program properties are trivially preserved.

When a new class is created, there are no instances of it, nor are there any subclasses that inherit from it. Therefore, the behavior of the program does not change when the new class is added.

B. *create_member_variable*

Add an unreferenced locally defined member variable to a class.

Arguments: variable V, class C.

Preconditions:

1. $\sim$varNameCollisionP(V.name, C)
   (the name of the new variable does not clash with an existing member or global variable).

The precondition ensures that the new variable doesn't collide with existing variables, satisfying program properties three (distinct member names), four (inherited Member Variables Not Redefined) and eight (semantically Equivalent References and Operations). Other program properties trivially preserved.

Adding a new, unreferenced variable does not change the behavior of a program.[1]

C. *create_member_function.*

Add a locally defined member function to a class that either is unreferenced or is identical to an already inherited function.

Arguments: function F, class C.

Preconditions:

1. $\forall$ memberFunction $\in$ C.locallyDefinedMemberFunctions,
   memberFunction.name $\neq$ F.name.
   (the function is not already defined locally)
2. $\forall$ F2 $\in$ inheritedMemberFunctions(C),
   (F2.name = F.name) $\Rightarrow$
   matchingSignatureP (F, F2).
   (the signature matches that of any inherited function with the same name)
3. $\forall$ F3 $\in$ functionsThatOverride(F),
   matchingSignatureP(F, F3).
   (the signatures of corresponding functions in subclasses match it)
4. $\forall$ F2 $\in$ inheritedMemberFunctions(C),
   (F2.name = F.name) $\Rightarrow$
   ($\forall$ class $\in$ C $\cup$ subclassesOf(C),
   unrefdOnInstancesP(F2, class)) $\vee$
   (semanticallyEquivalantP F, F2).
   (if there is an inherited function with the same name, either the inherited function is unreferenced on instances of C (and its subclasses), or the new function is semantically equivalent to the function it replaces)
5. compilesP(F, C).
   (F will compile as a member of C.)

[1]This refactoring increases the physical size of instances of the class. Programs that test the physical size of objects could see their behavior change (see the discussion in section 4.1.2).

Precondition one ensures that member names are distinct (property one). Preconditions two and three ensure that the member function has a signature identical to any corresponding superclass or subclass function. By precondition four, if the new function collides with an existing function, its function body must match the body of the function it is replacing, ensuring semantically consistent references and operations (property seven). Other program properties are trivially preserved.

The behavior of a class is not changed when a function is added that either is unreferenced or replaces an identically defined, inherited function.[2]

5.2 Deleting a Program Entity

These refactorings delete an unreferenced class or delete an unreferenced or redundant class member. Each refactoring is behavior preserving because the entity (class or class member) it deletes is either unreferenced or redundant. A member function is redundant if an identical member function would be inherited from a superclass if it were deleted.

A. *delete_unreferenced_class*

Delete an unreferenced class.

Arguments: class C.

Preconditions:

1. referencesTo(C) = $\emptyset \wedge$ subclassesOf(C) = $\emptyset$.
   (the class being deleted is unreferenced and has no subclasses).

The class being deleted from the program is unreferenced; thus, all program properties are trivially preserved.

B. *delete_unreferenced_variable*

Delete an unreferenced variable.

Arguments: variable V.

Preconditions:

1. referencesTo(V) = $\emptyset$.
   (the variable being deleted is unreferenced)

The variable being deleted from the class is unreferenced; thus, all program properties are trivially preserved.[3]

C. *delete_member_functions*

Delete a set of member functions from their class.

Arguments: setOf(function) functionsToDelete.

Preconditions:

[2]This refactoring increases the physical size of instances of the class. Programs that test the physical size of objects could see their behavior change (see the discussion in section 4.1.2).

[3]This refactoring decreases the physical size of instances of the class. Programs that test the physical size of objects could see their behavior change (see the discussion in section 4.1.2).

```
1. $\forall$ F $\in$ functionsToDelete:
       ($\forall$ ref $\in$ referencesTo(F):
           containingFunction(ref) $\in$
               functionsToDelete) $\vee$
       $\exists$ F2 $\in$
           (F.owner.superclass).locallyDefinedMemberFunctions $\vee$
               inheritedMemberFunctions(F.owner.superclass)
           $\wedge$ F2.accessControlMode $\neq$ private
           $\wedge$ matchingAttributesP(F, F2).
   (either each function being deleted is redundant, or the only references to it are by
   other functions that are also being deleted)
```

By the precondition, each function is either redundant and therefore can be deleted without changing the behavior of the program, or is unreferenced after the set of other functions are deleted. This preserves program property seven (semantically equivalent references and operations). Other program properties are trivially preserved.

5.3 Changing a Program Entity

These refactorings change the name of a class or change the attributes of a class member. They change a class member by changing its name, access control mode or type. Other changes include: converting an instance variable to a variable that points to an instance, converting a variable reference to a function call, and adding or reordering function arguments.

A. *change_class_name*

Change the name of a class.

Arguments: class C, string S.

This name change is reflected throughout the program (i.e. in class and subclass declarations, constructor and destructor functions, and the declarations of instances of the class).

Preconditions:

```
1. $\forall$ class $\in$ Program.classes,
       class.name $\neq$ S.
   (the new name doesn't clash with an already existing class)
```

The precondition ensures distinct class names (satisfying program property two). Changing the name of a class does not change its behavior (satisfying program property seven). Other program properties are trivially preserved.

B. *change_variable_name*

Change the name of a variable.

The name change is reflected throughout its scope. The variable could be a global variable, member variable, temporary variable or be an argument to a function. The rules of the language determine where a variable is visible.

Arguments: variable V, string newName.

Preconditions:

1. $\sim$varNameCollisionP(newName, scopeOf(V))
   (newName does not clash with an existing member or global variable).

The precondition ensures that the new variable doesn't collide with existing variables, satisfying program properties three (distinct member names), four (variable scoping) and five (member variable redefinition). Other program properties are trivially preserved.

The new variable name does not collide with a referenced or inherited variable. Changing the name of the variable does not change its behavior, as long as references to the variable remain the same.

C. *change_member_function_name*

Change the name of a member function and any corresponding member functions defined in subclasses (and callers).

Arguments: function F, string newName.

Preconditions:

1. $\forall$ member $\in$ F.owner.locallyDefinedMemberFunctions,
   member.name $\neq$ newName.
   (a function with the same name is not already defined locally)
2. if F.accessControlMode $\neq$ private,
   $\forall$ subClass $\in$ subclassesOf(F.owner),
   $\forall$ member $\in$ subClass.locallyDefinedMemberFunctions,
   member.name $\neq$ newName.
   (if F is not private, a function with same name is not already defined locally in a subclass)
3. $\forall$ F2 $\in$ inheritedMemberFunctions(containingClass(F)),
   (F2.name = F.name) $\Rightarrow$
   matchingSignatureP (F, F2).
   (the signature matches that of any inherited function with the same name)
4. $\forall$ F2 $\in$ inheritedMemberFunctions(containingClass(F)),
   (F2.name = F.name) $\Rightarrow$
   ($\forall$ class $\in$ containingClass(F) $\cup$ subclassesOf(containingClass(F)),
   unrefdOnInstancesP(F2, class)) $\vee$
   (semanticallyEquivalantP F, F2).
   (if there is an inherited function with the same name, either the inherited function is unreferenced on instances of containingClass(F) (and its subclasses), or the new function is semantically equivalent to the function it replaces)
5. $\forall$ F2 $\in$ inheritedMemberFunctions(containingClass(F)),
   F2.name = F.name $\Rightarrow$
   matchingSignatureP (F, F2) $\wedge$
   ($\forall$ ref $\in$ referencesTo(F2),
   containingClass(ref) = containingClass(F2)) $\vee$

($\forall$ class $\in$ (containingClass(F)) $\cup$ subclassesOf((containingClass(F))),
instancesOf((containingClass(F))) = $\emptyset$) $\vee$
(semanticallyEquivalantP F, F2).

(if there is an inherited member function with the same name, the signatures match. Also, either the inherited function is unreferenced on instances of the class that contains F (or its subclasses), or the renamed function is semantically equivalent to the function it replaces.)

The first three preconditions ensure that member names are distinct (property one). Precondition three ensures compatible signatures (property six; member function redefinition). By precondition four, if there is an inherited function with the same name, either the inherited function is unreferenced on instances of containingClass(F) and its subclasses, or the new function is semantically equivalent to the function it replaces, ensuring semantically consistent references and operations (property seven). Other program properties are trivially preserved.

D. *change_type*

Change the type of a set of (pointer) variables and functions (i.e. change the types of the variables and the return types of the functions).

Arguments: setOf(variable) varsToChange, setOf(function) functionsToChange,
type newType.

Preconditions:

1. $\forall$ Var $\in$ varsToChange,
   $\forall$ expression $\in$ expressionsAssignedTo(variableToChange):
   subtypeP(expression.type newType) $\vee$
   expression $\in$ (varsToChange $\cup$ functionsToChange).

   (each assignment *to* a variable would remain type safe if its type was changed to newType).

2. $\forall$ I $\in$ (varsToChange $\cup$ functionsToChange),
   $\forall$ expression $\in$ variablesAssigned(item):
   subtypeP(newType, expression.type) $\vee$
   expression $\in$ varsToChange.

   (each assignment *of* item would remain type safe if its type was changed to newType).

Preconditions one and two ensure type safe assignments (property six). Other program properties are trivially preserved.

E. *change_access_control_mode*

Change the access control mode of a member variable or function.

Arguments: (variable + function) member,
('private' OR 'protected' OR 'public') newAcMode.

Preconditions:

1. if (newAcMode = private) $\wedge$ (member.accessControlMode $\neq$ private),
   classesInternallyReferencing(member) = $\{containingClass(member)\}$ $\wedge$

classesPubliclyReferencing(member) $= \emptyset$.
(if the new mode is private, the member is only (internally) referenced, in the class where it is defined)

2. if (newAcMode $\neq$ private) $\wedge$ (member.accessControlMode = private)
   (a) if member is a variable:

   ```
   (∀ subclass ∈ subclassesOf(member.owner),
       ∀ Var ∈ subclass.locallyDefinedMemberVariables,
           Var.name ≠ member.name).
   ```

   (a member variable to be declared protected or public must not already be defined in a subclass)

   (b) if member is a function:

   ```
   i. ∀ F2 ∈ inheritedMemberFunctions(C),
          (F2.name = F.name) ⇒
              (∀ class ∈ C ∪ subclassesOf(C),
                  unrefdOnInstancesP(F2, class)) ∨
              (semanticallyEquivalantP F, F2).
   ```

   (if there is an inherited function with the same name, either the inherited function is unreferenced on instances of C (or its subclasses), or the new function is semantically equivalent to the function it replaces)

   ```
   ii. ∀ subclass ∈ subclassesOf(member.owner),
           ∃ F ∈ subclass.locallyDefinedMemberFunctions ∧
               F.name = member.name ⇒
                   matchingSignatureP(F, member).
   ```

   (the signature of the new function matches the signatures of functions that would override it in subclasses)

3. if (newAcMode = protected) $\wedge$ (member.accessControlMode = public),
   classesPubliclyReferencing(member) $= \emptyset$.
   (a member currently declared public but about to be declared protected must only be (internally) referenced, in the class that contains it or in one of its subclasses).

All references to the member before the refactoring are either within the member's scope after the refactoring, or (if a function) may be replaced by a reference to a semantically equivalent function, preserving semantically equivalent references and operations (property seven). Other program properties are trivially preserved.

F. *add_function_argument*

Add a new argument to the definition of a function in its containing class (and to functions that override it in subclasses). In each function call, add an argument which is a dynamically created instance of the type of newArg.

This refactoring is used, for example, when making the signatures of functions in different subclasses match.

Arguments: function F, variable newArg, newArg.type defaultValue[4].

Preconditions:

[4] This argument can be omitted if there are no calls to F.

1. $\forall$ Func $\in$ (F $\cup$ functionsThatOverride(F)),
   ~varNameCollisionP(newArg.name containingClass(Func)).
   (the name of newArg doesn't collide with another variable in an enclosing scope)
2. $\forall$ Func $\in$ memberFunctions(F.owner.superclass),
   Func.name $\neq$ F.name.
   (a member function with the same name is not defined in a superclass).
3. classesReferencing(F) $\subset$ scopeOf(defaultValue).
   (defaultValue is visible in all places where F is called).

By the precondition 1, there are no name collisions with the new variable in F, nor in the corresponding functions defined in subclasses. By preconditions 2, this covers all corresponding functions. The argument is unreferenced, hence property seven (semantically equivalent references and operations) is preserved. The other program properties are trivially preserved. Adding an unreferenced argument does not change the behavior of the program.

G. *delete_function_argument*

Delete an argument from the definition of a function in its containing class and, unless it is private, from all subclasses where it is redefined. In each function call, the argument is removed.

Arguments: variable Arg

Preconditions:

1. referencesTo(Arg) = 0.
   (the argument is unreferenced.)
2. $\forall$ F $\in$ (functionsThatOverride(Arg.owner)),
   referencesTo(argument in corresponding position) = 0.
   (the corresponding argument in each overriding function is unreferenced).
3. $\forall$ Func $\in$ memberFunctions(Arg.owner.owner.superclass),
   Func.name $\neq$ Arg.owner.name.
   (the member function containing Arg is not defined in a superclass).
4. $\forall$ argExpr assigned to Arg (or to the corresponding argument in each overriding function),
   noSideEffectsP(argExpr).
   (the expressions passed to Arg (or to corresponding argument variables in overriding functions) do not have side effects).

By the preconditions, Arg is not referenced, nor is the corresponding argument referenced in overriding functions defined in subclasses. Expressions passed to Arg (in calls to its containing function) have no side effects. Therefore, program property seven (semantically equivalent references and operations) is preserved. The other program properties are trivially preserved. Deleting an unreferenced argument does not change the behavior of the program.

H. *reorder_function_arguments*

Reorder the arguments in a member function definition (in the specified class and in all subclasses) and in all calls to that function.

Arguments: function F, (list of variable) newArgOrdering.

Preconditions:

1. $\forall$ function $\in$ (F $\cup$ functionsThatOverride(F)),
   $\forall$ argument $\in$ function.arguments
   $\forall$ expression $\in$ argumentExpressionsInFunctionCalls(argument, F):
   noSideEffectsP(expression).
   (the expressions assigned to the argument variables have no side effects)[5]
2. correspondingSuperclassMemberFunctions(F) = $\emptyset$.
   (a corresponding member function is not defined in a superclass).

The function arguments (by precondition) do not have any side effects; reordering them does not change the behavior of the function, since the ordering of arguments in all calls to the function are similarly reordered. Thus, program property seven (semantically equivalent references and operations) preserved by the preconditions; other program properties are trivially preserved.

I. *add_function_body*

Add a function body to an existing member function (signature).

Arguments: function F, statement newFunctionBody.

Preconditions:

1. compilesP (F, where F.body = newFunctionBody), containingClass(F))
   (the new function body compiles).

By precondition one, the function body compiles cleanly. Since the function is a signature only, it is never called directly; adding a function body to it does not change the behavior of the program. Therefore program property seven (semantically equivalent references and operations) is preserved, as are (trivially) the other program properties.

J. *delete_function_body*

Delete a function body from an existing member function (making it a signature only).

Arguments: function F.

Preconditions:

1. callsTo(F) = $\emptyset$.

By the precondition, F is never called in the program, therefore behavior is preserved when its function body is removed. Therefore, program property seven (semantically equivalent references and operations) preserved, as are (trivially) the other program properties.

[5]If two argument expressions in a function call have side effects, and the side effects are not independent, program behavior could change if the arguments were reordered. This precondition prevents this from happening. The precondition could be relaxed to permit only one argument expression with side effects per function call, or multiple argument expressions with side effects as long as the side effects are independent of each other.

K. *convert_instance_variable_to_pointer.*

This refactoring is specific to C++. C++ has two ways in which a variable *Var1* can "contain" an instance of a class Class1. The variable declaration "Class1 *Var1*" declares *Var1* to be a particular instance of Class1 throughout program execution. Space is statically allocated for it at compile time. *Var1* can be reassigned, but can only be reassigned an instance of Class1.

By contrast, the variable declaration "Class1* *Var1*" allows *Var1* to point to (i.e. contain) an instance of Class1 or an instance of any of its subclasses. This allows expressions assigned to it to be specialized while maintaining type compatibility.

This refactoring converts a variable from its current type to become a pointer to that type, and assigns to the variable an instance of its current type.

If the variable is a member variable, with a declaration of the form "Class1 *Var1(args)*", this refactoring changes the declaration to "Class1* *Var1*", adds to the class constructor the assignment: "*Var1 = new* Class1*(args)*", and adds to the class destructor the statement: "*delete Var1*".

If the variable is a local variable, with a declaration of the form "Class1 *Var1(args)*", this refactoring changes the declaration to "Class1* *Var1*" and immediately following the variable declaration adds the assignment: "*Var1 = new* Class1*(args)*".

Expressions of the form *&Var1* are changed to *Var1*. Expressions of the form *Var1.member* are changed to *Var1− >member*.

Arguments: variable member_variable.

Preconditions:

1. none.

Program property six (type safe assignments) is preserved by construction. Also, by construction each reference to the variable is replaced by a semantically equivalent (but not identical[6]) reference to a pointer variable, preserving property seven. Other program properties are trivially preserved.[7]

L. *convert_variable_references_to_function_calls*

Arguments: variable V.

Convert all references to V to calls to its accessing functions. Convert all assignments to V to calls to its updating function.

Preconditions:

1. locallyDefinedMemberNamedP(get_$< V >$, V.owner) $\wedge$
   locallyDefinedMemberNamedP(address_of_$< V >$, V.owner) $\wedge$
   locallyDefinedMemberNamedP(set_$< V >$, V.owner).
   (the access and update functions are already defined).

[6] See section 8.5.3 for a discussion of how variable references can be semantically equivalent but not identical.

[7] This refactoring can decrease the physical size of instances of the class. Programs that test the physical size of objects could see their behavior change (see the discussion in section 4.1.2).

The access and update functions, by construction, behave the same as direct accesses and updates to the variable. Therefore, replacing the variable accesses with calls to these functions preserves property seven (semantically equivalent references and operations). Other program properties are trivially preserved.

M. *replace_statement_list_with_function_call*

Arguments: function F, list statementList, functionCall FC.

Replace statementList in F with the function call FC.

Preconditions:

1. classContaining(F) $\subset$ scopeOf(FC.calledFunction)
   (the called function is visible from the calling function).
2. semanticallyEquivalentP(statementList, FC).
   (the function call is semantically equivalent to the statement list it replaces. The abstract syntax trees must be the same, up to variable renaming.)

The function call, by precondition, behaves the same as the statement list it replaces. Program property seven (semantically equivalent references and operations) is therefore preserved; other program properties are trivially preserved.

N. *inline_function_call*

Replace a function call with the body of the called function.[8]

Arguments: functionCall FC.

Preconditions:

1. $\forall$ refdItem $\in$
   (variablesReferencedBy(FC.calledFunction) $\cup$
   functionsCalledBy(FC.calledFunction)),
   (scopeOf(refdItem) $\subset$ FC.calledFunction) $\vee$
   (containingClass(FC) $\subset$ scopeOf(refdItem)).
   (all member variables and functions referenced by the called function are reachable from the calling function)

After checking the preconditions, this refactoring:

1. creates a statement list, equivalent to the function body in the called function, to insert into the calling function. In creating the statement list, the names of variables local to the called function may be changed to avoid name collisions. Each call by value is handled by assignment statement added to the beginning of the inlined code. Return statements are converted to branch/goto statements (at the end of the statement list).
2. replaces the function call with the equivalent statement list.

[8]Issues involved in inlining functions calls are detailed in the program transformation literature [27].

The only problem with inlining a called function is that the function may reference private or protected members that the calling function can't reference. By precondition one, all references are visible to the caller. Thus, the statement list is semantically equivalent to the function call it replaces. Therefore, program property seven (semantically equivalent references and operations) is preserved. Other program properties are trivially preserved.

O. *change_superclass*

Change the superclass of a class.

Arguments: class C, class newSuperclass.

Preconditions:

1. $\forall$ I $\in$ instancesOf(C $\cup$ subclassesOf(C)),
   $\forall$ Var $\in$ variablesAssigned(I),
   Var.type $\in$ (C $\cup$ subclassesOf(C) $\cup$ newSuperclass $\cup$
   superclassesOf(newSuperclass)).
   (each assignment *of* item would remain type safe if its type was changed to new_type).
2. $\forall$ member $\in$ inheritedMembers(C),
   memberNamedP(member.name, newSuperclass) $\wedge$
   matchingAttributesP(
   member,
   memberNamed(member.name, newSuperclass))
   (all members currently inherited will be identically inherited from the new superclass. This precondition is strict, but the higher level refactoring move_class satisfies this precondition by first calling other refactorings to add members as needed to the superclass, before calling this refactoring.)
3. $\forall$ member $\in$ locallyDefinedMembersOf(C),
   if memberNamedP(member.name, newSuperclass),
   (if member is a variable): matchingAttributesP(
   member, memberNamed(member.name, newSuperclass))
   (if member is a function): matchingSignatureP(
   member, memberNamed(member.name, newSuperclass)).
   (each locally defined member either doesn't clash with a member inherited from the new superclass, or it is identical to an member inheritable from the new superclass and therefore can be removed from the subclass when the superclass is changed.)
4. the constructor and destructor chain for the new superclass is semantically equivalent to the chains for the current direct superclass of C.

After checking the preconditions, this refactoring:

1. changes the superclass attribute of C to newSuperclass
2. deletes from C any member variables identically defined in or inherited by newSuperclass.

Program property six (type-safe assignments) is preserved by precondition one. By preconditions two and three, references to members before the refactoring will be the same, or consistently replaced by references to members with an identical type, after the refactoring. By precondition four, the constructors and destructors are semantically equivalent when the superclass is changed. Therefore, program property seven (semantically equivalent references and updates) is preserved. Other program properties are trivially preserved.

5.4 Moving a Member Variable

These refactorings move a member variable to a superclass or subclass.

A. *move_member_variable_to_superclass*

Arguments: variable V, class C.

Move V to class C from all subclasses where it is defined. If V.accessControlMode = public in its current class(es), it will be public in its new class; otherwise, the access control mode in the new class will be protected. Also, in each subclass where it is currently defined, the local definition of the variable is removed, as it will be now be inherited from C.

This refactoring changes the relative physical ordering of variables in the class, but the variable type and references remain the same.[9]

Preconditions:

1. $\forall$ C $\in$ subclassesOf(C):
   ($\exists$ Var $\in$ C.locallyDefinedMemberVariables $\wedge$
   Var.name = V.name) $\Rightarrow$
   matchingAttributesP(Var, V).
   (the variable is defined identically in all subclasses where it is defined)
2. $\forall$ member $\in$ C.locallyDefinedMemberVariables,
   member.name $\neq$ V.name)
   (the variable isn't already defined locally in (as a private member of) the superclass).

Since (by precondition one) the variable is defined identically in all subclasses where it is defined, replacing the local definitions in the subclasses by a definition inherited from C will not change the behavior of the program, preserving program property seven (semantically equivalent references and operations). Other program properties are trivially preserved.[10]

B. *move_member_variable_to_subclasses*

Arguments: variable V, subclass_list.

Move the member variable from its current containing class to each of its immediate subclasses.

Preconditions:

[9] This refactoring changes the relative physical position of variables within instances of the class. Programs that test the physical positions of variables could see their behavior change (see the discussion in section 4.1.2).

[10] This refactoring changes the relative physical position of variables within instances of the class. Programs that test the physical positions of variables could see their behavior change (see the discussion in section 4.1.2).

1. referencesTo(V, containingClass(V)) = $\emptyset$
   (the variable is not referenced by members of, nor referenced on instances of, containingClass(V))
2. V.accessControlMode $\neq$ private.
   (the variable is already inherited by the immediate subclasses.

After the refactoring, V will still be defined in all subclasses that currently reference it. Therefore, program property seven (semantically equivalent references and operations) is preserved by preconditions; other program properties are trivially preserved.[11]

5.5 Composite Refactorings

These refactorings build upon the other low-level refactorings, to: abstract variable access, convert between a code segment and a corresponding function call, and perform more powerful moving operations.

A. *abstract_access_to_member_variable*

Abstract access to a member variable by defining functions for accessing the value or address of the variable, and for updating its value. Replace variable references with calls to these functions, and (if the access control mode of the variable is public) limit access to the variable by making it protected.

Arguments: variable V.

After checking its preconditions, this refactoring:

1. calls create_member_function to create the access and update functions
   (named get_$< V >$, set_$< V >$, address_of_$< V >$)
2. calls convert_variable_references_to_function_calls to replace references to V with calls to the access and update functions.
3. if the access control mode of V is public, calls change_access_control_mode to make V protected.

Preconditions:

1. ~memberNamed(get_$< V >$ containingClass(V)) $\wedge$
   ~memberNamed(set_$< V >$ containingClass(V)) $\wedge$
   ~memberNamed(address_of_$< V >$ containingClass(V)).
   (access to the variable has not already been abstracted)

For create_member_function: preconditions one through four are satisfied by precondition one of this refactoring; precondition five is satisfied because the new functions only contain a reference to V, which remains visible within its class. For the refactoring convert_variable_references_to_function_calls: precondition one is satisfied by the functions were created in the prior step. For change_access_control_mode: since this refactoring is

[11]This refactoring changes the relative physical position of variables within instances of the class. Programs that test the physical positions of variables could see their behavior change (see the discussion in section 4.1.2).

only called if V is public, preconditions one and two are trivially satisfied; precondition three is satisfied because after the prior step, the only reference to the variable is from functions defined in the class.

This refactoring is constructed using behavior preserving refactorings and, since it satisfies the preconditions of those refactorings, behavior is preserved.

B. *convert_code_segment_to_function*

Define a new member function, with a unique name, whose function body is equivalent to a statement list of an existing member function. Replace that statement list in the existing member function with a call to the new member function.

Arguments: setOf(statement) statementList.

After checking its precondition, this refactoring:

1. constructs an argument list for the new member function. The argument consists of variables local to containingFunction(statementList) that are referenced or modified by statementList, but which are declared within statementList. Variables modified is statementList will be passed-by-referenced; other variables will be passed-by-value.

   Then, calls create_member_function to create the new member function. The new function is given a unique name.
2. calls replace_statement_list_with_function_call to replace the statement list with a call to the newly created function.

Preconditions:

1. convertibleToFunctionP(statementList).
   (statementList is convertible to a legal function)

For create_member_function: preconditions one through four are satisfied because the function is given a unique name; precondition five is satisfied because the new function has the same access to global and member variables and functions as the old function, and any variables local to the old function but not defined in statementList are passed as arguments to the new function. For replace_statement_list_with_function_call, precondition one by the results of the prior step; precondition two is satisfied because, by construction, the new function is equivalent to the statement list it replaces.

This refactoring is constructed using behavior preserving refactorings and, since it satisfies the preconditions of those refactorings, behavior is preserved.

C. *move_class*

Move a class (change its superclass).

Arguments: class C, class newSuperclass.

After checking the preconditions, this refactoring:

1. determines the set of inherited members that would not be (identically) inherited by C (from newSuperclass or one of its superclasses) at its new location[12]; then:

[12]The current subclass of C could share a common superclass with newSuperclass; these classes may inherit some common behavior but a virtual member function could be defined differently in those classes. Or, the

   (a) for each member function in the set, calls create_member_function to copy the function into C
   (b) for each member variable in the set:
      i. if the variable is defined in any current subclass of newSuperclass, calls move_member_variable_to_superclass to move the variable to newSuperclass.
      ii. otherwise if the variable is not already a local or inherited member of newSuperclass, calls create_member_variable to add the variable to newSuperclass.
2. calls the refactoring change_superclass to change the superclass of C.

Preconditions:

1. $\forall$ member $\in$ (membersOf(C):
   (a) if member is a variable $\wedge$
         $\exists$ Var $\in$ memberVariablesOf(newSuperclass) $\wedge$
         member.name = Var.name,
            matchingAttributesP(member, Var).
   (b) if member is a function $\wedge$
         $\exists$ Func $\in$ memberFunctionsOf(newSuperclass) $\wedge$
         member.name = Func.name,
            matchingSignatureP(member, Func).
   (attributes of member variables and signatures of member functions in C will match with its superclasses.)
2. if variables needed in the new superclass are already defined in its subclasses, those variables are defined identically in the subclasses
3. $\forall$ subclassInstance $\in$ instancesOf(C),
      $\forall$ Var $\in$ variablesAssigned(subclassInstance),
         Var.type $\in$ (C $\cup$ newSuperclass
            $\cup$ superclassesOf(newSuperclass)).
   (each assignment *of* an instance of the class being moved would remain type safe).

For create_member_function: preconditions one and four are satisfied because the refactoring is only invoked if it is not already locally defined or inherited; precondition two is satisfied by precondition one of this refactoring; precondition three is satisfied because the class is moved along with its superclasses; precondition five is satisfied by construction - this refactoring ensures that all referenced variables and functions are visible after the move.

For move_member_variable_to_superclass: precondition one is satisfied by precondition two of this refactoring; precondition two is satisfied by this refactoring is not invoked when the variable is already defined in or inherited by the superclass.

For create_member_variable: this refactoring is only invoked if the variable does not collide with an existing variable, thus satisfying precondition one.

current and new superclasses may not share a common superclass and all inherited behavior would need to be copied.

For change_superclass: precondition one is satisfied by precondition three of this refactoring; precondition is satisfied by the results of the prior steps; precondition three is satisfied by precondition one of this refactoring.

This refactoring is constructed using behavior preserving refactorings and, since it satisfies the preconditions of those refactorings, behavior is preserved.[13]

[13] This refactoring could change the physical size of instances of the class. Programs that test the physical size of objects could see their behavior change (see the discussion in section 4.1.2).

Chapter 6

Refactoring To Generalize: Creating an Abstract Superclass

This chapter focuses on creating an abstract superclass that captures the behavior common to two classes. It can be generalized to work with any number of classes.

Casais [30] and Bergstein [15] note that a useful behavior preserving program transformation is to move a function or variable to a common superclass from subclasses where it is identically defined. However, as this chapter describes defining an abstract superclass involves more than just moving up identically defined members. The important (and complex) parts of the process come in finding the common abstraction that is often hidden behind differing vocabulary, and achieving compatibility between the subclass implementations.[1]

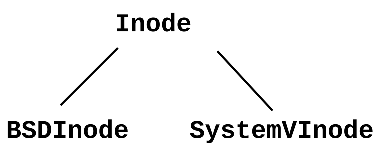
When capturing the abstraction common to two classes, one cannot expect that the similarities will already be obvious, conveniently separated from the differences, and defined identically in both subclasses. In general, achieving compatibility between the subclasses may require changing member names and other attributes, and splitting function bodies, before members can be moved up to the superclass.

To illustrate some of the complexities, recall the example described in section 3.2 for creating the abstract superclass Inode, that captured the abstraction common to the BSDInode and SystemVInode classes. The common abstraction was contained in functions that were *not* identically defined in the two classes before the refactoring was performed. In particular, the implementations of the *mapUnit* function were slightly different due to differences in the two file formats.

As shown below in Figure 6.1, most but not all of the code for the *mapUnit* function in the SystemVInode class was the same as for the *mapUnit* function defined in the BSDInode class. However, the code for getting and setting a logical block number was different between the two file system formats.

The implementations of the *mapUnit* function in each subclass needed to match before they could be replaced by a commonly inherited function in the superclass. To handle this, the code that differed between the two implementations was placed into new functions *getDirect* and *setDirect* which were added to the subclasses. Then, in both definitions of the *mapUnit*

[1]Casais [31] similarly observes that renamings and other attribute refinements are sometimes needed before functions and variables are migrated to an abstract superclass.



BEFORE:

```
BSDInode

   Protected:

      int  mapUnit(...)  {
              *
         code to get
         physical block #
         (BSD specific)
              *
         code to set
         physical block #
         (BSD specific)
              *           }

SystemVInode

   Protected:

      int  mapUnit(...)  {
              *
         code to get
         physical block #
         (SystemV specific)
              *
         code to set
         physical block #
         (SystemV specific)
              *           }
```

AFTER:

```
Inode

   Protected:

      virtual int getDirect()
      virtual void setDirect(..)
      int  mapUnit(...)  {
              *
         index = getDirect()
              *
         setDirect(index)
              *           }

BSDInode  (and SystemVInode)

   Protected:

      int  getDirect() {...}
      void setDirect(..) {...}
```

Figure 6.1: Refactoring The Function *mapUnit*

function, the differing code was replaced by calls to these functions. Once the definitions of the *mapUnit* function in both subclasses matched, the function was moved to the superclass.

In general, the process of defining an abstract superclass first involves creating a new class and moving the two existing classes under it. Then, capturing the common abstraction in the superclass is an iterative process that may involve the following steps:

- making function signatures compatible in both subclasses
- adding function signatures to the superclass protocol
- making functions bodies (and the variables referenced by them) compatible in both subclasses
- migrating common code and variables to the superclass.

Each of these steps is described in this chapter. The chapter closes with a discussion of how interaction with the designer might be reduced, and other issues.

Casais [31] deals with several of the issues discussed in this chapter: renamings, refining function signatures and variable attributes, and moving members. One important step defined here that is not addressed by Casais [31] is how to separate out the differences between corresponding functions defined in both subclasses, making them match, before migrating common code to the superclass. That step closely relates with the other steps defined in this chapter. There are other ways in which this work is different from [31]: it provides some heuristics for detecting structural similarities, includes access control mode as an attribute that can be refined, and discusses some of the tradeoffs involving user/tool interaction.

Before a function signature or variable can be added to the superclass, it must first be defined compatibly in the subclasses. Compatibility is defined differently for the various object-oriented languages and programming styles. The refactorings defined in this chapter require that:

- for function signatures to be compatible the names, argument types, return types and access control modes must match
- for variables to be compatible the names, types and (for member variables) access control modes must match.

For each of the refactorings described in this and the following two chapters, the description of the refactoring is followed by sections that detail its preconditions and argue that it is behavior preserving. As with the low-level refactorings defined in the previous chapter, these sections (included for completeness) provide a level of detail beyond what may be of interest to some readers. These sections can be skipped without missing the intent of the refactorings.

6.1 Creating a Common Superclass

The first important step in this process (before dealing with function signatures, common code and variables) is to create the new abstract superclass. It is assumed here that the two classes that share a common abstraction have already been selected by the designer[2] and are *sibling* classes; that is, they either share a common direct superclass or each class is the top most class in its inheritance hierarchy. If the two classes are not sibling classes, the *move_class* refactoring, described in chapter five, can be applied to move one or both of the classes.

The arguments to this refactoring are:

- string absSuperclassName
- class C1, C2.

After checking its preconditions, this refactoring:

1. calls the create_empty_class refactoring to create the new superclass, as a sibling of the two current classes.
2. calls the move_class refactoring to move the two classes under it.

[2]Section 6.8.2 discusses issues involved in automatically selecting two classes with common attributes from a set of classes.

6.1.1 Preconditions

The preconditions for this refactoring are:

1. $\forall$ class $\in$ Program.classes,
   class.name $\neq$ absSuperclassName.
   (absSuperclassName does not clash with the name of an existing class)

2. C1.superclass = C2.superclass
   (the two classes share a common superclass).

6.1.2 Behavior Preservation

For create_empty_class, precondition 1 (the name of the new class doesn't clash with an already existing class) is satisfied by precondition 1 of this refactoring.

For move_class: The preconditions are satisfied because after the move classes S1 and S2 will inherited the same members as before (satisfying precondition 1), no members need to be locally added to the classes (satisfying precondition 2) and the superclasses (subclasses) of C1 and C2 prior to the refactoring will continue to be superclasses (subclasses) of C1 and C2 after the refactoring (satisfying precondition 3).

This refactoring is constructed using behavior preserving refactorings and, since it satisfies the preconditions of those refactorings, behavior is preserved.

6.2 Making Function Signatures Compatible

Functions belong in the superclass protocol if they are part of the common abstraction represented by the superclass. As noted in chapter one, determining the correct abstraction is a design decision that cannot be entirely automated. Heuristics (such as those listed below) can detect similarities between classes, but they are not foolproof.

Two subclasses sharing a common abstraction may have member functions that are similar but not exactly the same. This might be expected especially if the two classes were developed independently of each other. The name of the function in one class may be a synonym of the function defined in the other class. Or, the functions may have the same names, but the return types or function arguments may be slightly different. In these cases, the signatures of the member functions in the subclasses need to be changed before they match with each other.

This does not imply, however, that detecting common abstractions must be left entirely to the user. It is possible to automatically infer design similarities from structural similarities, perform some primitive refactorings that make the functions more structurally similar, and add all possible matches to the protocol of the abstract class. There are at least two shortcomings to carrying automated support to this extreme:

1. A tool might detect and act upon structural matches that do not correspond to meaningful common abstractions. The vocabulary used to name classes and class members is often ambiguous. Terms such as length, size, age, and brightness can have different meanings in different classes, and variables with these names may not belong in a common abstraction even though they are defined in both classes.

   If the two classes that share a common abstraction were developed independently, common members may have different names. This suggests checking for matches on attributes other than name. Unfortunately, members with similar attributes need not represent similar abstractions. For example, suppose an abstract class Vehicle is created from the existing classes Automobile and Submarine. Suppose that each class contained a function with a single, integer argument. These functions are structurally similar, and an automated tool could detect this similarity, rename one of the functions and define a common function in the superclass. But, suppose that in the Automobile class, the function is called *changeOil* whose argument is the number of quarts of oil needed; in the Submarine class, the function is called *submerge* whose argument is the depth to which to descend. These functions clearly don't share a common abstraction, but automatically matching on attributes won't detect this.

2. Some of the structural matches may collide with other matches. Suppose a function in one class may have a return type and argument types that match two or more functions in the other class, all with different names. In this case, it is not clear which of several matches should be added to the superclass. This implies that, before selecting among these possible matches, it is necessary to check how these functions are used.

Thus, automating the detection of a common abstraction is possible but has its shortcomings. These factors suggest a middle ground, where a refactoring tool displays alternative refactorings and lets the user select the one to perform, as shown in Figure 6.2.

In a multi-window display presented to the user, a refactoring tool could list the functions in the superclass protocol, and list the functions defined in each subclass. In this example, a function is defined in both the Automobile and Submarine classes for redirecting the vehicle. In one class, the function is called *shiftDirection* while in the other class it is called *redirect*. The arguments to the two functions are the same, but are in inverted order.

The refactoring tool could generate its set of suggested changes based on the structural similarities it detects between the classes. The tool could suggest renaming the function in one of the classes and reordering the arguments of one function. Once these changes are made, the signatures will be identical and the signature can be added to the superclass protocol. The tool could suggest possible refinements in the lowest window. For example, in comparing two functions in different classes, the return types and arguments may be the same but the function names may be different; the tool would suggest renaming one function to match the other. Below is a set of heuristics that can detect some structural similarities and suggest refinements. For example, heuristic 2 detects the case where the a function in one subclass has the same arguments and return types, as a function (with a different name) defined in the other subclass. These structural similarities suggest that the two functions

These heuristics can be applied, repeatedly, to iteratively derive a superclass protocol. First, the most obvious case is checked, where a function is identically defined in both classes. When this occurs, the heuristic suggests adding the function signature to the superclass protocol. In other cases, the attributes of a function in one class will be similar to but not exactly the same as an equivalent function in the other class. The other heuristics below handle renaming (2-4), retyping (5-6), reordering arguments (7) and changing access control mode (8), to make the signatures of equivalent functions match[3]:

[3]The action clauses of most heuristics suggest applying a primitive refactoring, described in chapter five. The preconditions of those refactorings are somewhat more complex than the preconditions shown here.

.

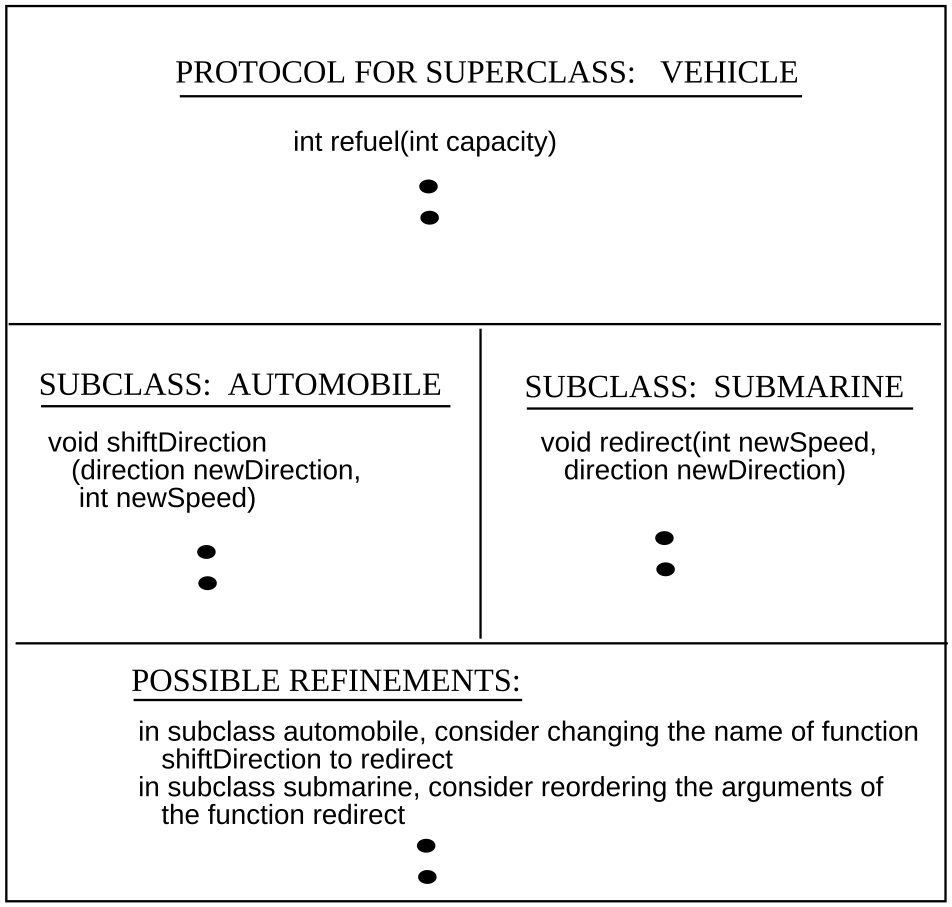

PROTOCOL FOR SUPERCLASS: VEHICLE
int refuel(int capacity)
SUBCLASS: AUTOMOBILE
void shiftDirection (direction newDirection, int newSpeed)
SUBCLASS: SUBMARINE
void redirect(int newSpeed, direction newDirection)
POSSIBLE REFINEMENTS:
in subclass automobile, consider changing the name of function shiftDirection to redirect
in subclass submarine, consider reordering the arguments of the function redirect

Figure 6.2: Defining the Superclass Protocol

```
1. if   function F1 is defined in class A
        function F1 is defined in class B
        the functions have the same access control mode, return type, and
          arguments with the same names and types
   then consider adding function F1 to the superclass protocol
     (using the refactoring defined in section 6.3).

2. if   function F1 is defined in class A
        function F1 is not defined in class B
        function F2 is defined in class B
        function F2 is not defined in class A
        functions F1 and F2 have arguments with the same names and
```

```
          types, and have the same return type
     then consider renaming function F2 in class B to F1
       (using the refactoring change_member_function_name).

  3. if   function F1 is defined in class A
          function F1 is not defined in class B
          function F2 is defined in class B
          function F2 is not defined in class A
          the arguments in function F2 of class B can be arranged such
            that the corresponding arguments in both functions have
            types that are the same, parent/child or siblings
     then consider renaming function F2 in class B to F1
       (using the refactoring change_member_function_name).

  4. if   function F1 is defined in class A
          function F1 is defined in class B
          the types of their corresponding arguments and return value
            are the same.
     then consider changing (as needed) the argument names in function F1
       of class B to match the argument names in class A,
       (using the refactoring change_variable_name).

  5. if   function F1 is defined in class A
          function F1 is defined in class B
          a type of an argument (or return value) in one function
            is the subtype to the type of the corresponding argument
            (or return value) in another function
     then consider changing child-type to parent-type
       (using the refactoring change_type).

  6. if   function F1 is defined in class A
          function F1 is defined in class B
          a type of an argument (or return value) in one function
            shares a common supertype with the type of the corresponding
            argument (or return value) in another function
     then consider changing both types to be their common ancestor
       (using the refactoring change_type).

  7. if   function F1 is defined in class A
          function F1 is defined in class B
          the arguments in function F1 of class B can be arranged such
          that the corresponding arguments in both functions have types
          that are the same, parent/child or siblings
     then consider reordering the arguments of F1 in class B
       (using the refactoring reorder_function_arguments).
```

```
8. if   function F1 is defined in class A
        function F1 is defined in class B
        their access control modes are different
   then consider changing the least permissive mode to match
     the most permissive mode,
     (using the refactoring change_access_control_mode).
```

Heuristics such as these could be applied incrementally to make the functions conform, or they could be chained together to find all possible matches.

For each heuristic, the name of the corresponding refactoring is shown in parentheses. This is done here to make clear the relationships between the suggested refinements and primitive refactorings. However, in the user interface (shown in Figure 6.2), the refactoring names are not displayed. Rather, when the user selects an attribute to update and enters a new value, the corresponding refactoring is automatically triggered. Such an update by the user may be in response to a refinement suggested by the tool, or might be a refinement (such as a renaming) based on the user's design insight. Before the primitive refactoring is performed, whose preconditions are checked. If the change is valid, the refactoring is performed, and the user display is updated. When the function signatures match, the user selects to add the function signature to the superclass protocol, in which case the refactoring described in the next section is invoked.

These heuristics detect similarities based on a small set of structural attributes. More powerful similarity detection is possible, as discussed in the artificial intelligence machine learning literature on analogy-based systems [28, 39, 51, 64]. Analogy-based systems have very limited power unless their analysis is based on a well-defined domain model. Such approaches are of dubious value in many applications of refactoring, where the design of the application is not well understood and is evolving.

6.3 Adding Function Signatures to the Superclass Protocol

Once the signature of a function in both subclasses match, the function signature can be added to the superclass. The following refactoring adds a function signature to the superclass protocol. The arguments to this refactoring are:

- the member function (F1) as defined in the first subclass (C1)
- the member function (F2) as defined in the second subclass (C2)

After checking its preconditions, this refactoring:

1. calls the create_member_function refactoring to add the function (signature) as a protected member of the superclass. The function is defined as virtual in the superclass (referred to below as 'commonSuperclass'), allowing it to be (re)defined in the subclasses.[4]

[4]In C++, the function can be defined as a pure virtual function by assigning its value to be '0' in the superclass.

6.3.1 Preconditions

The precondition for this refactoring is:

1. $\forall$ member $\in$ commonSuperclass.locallyDefinedMemberFunctions,
   member.name $\neq$ F1.name.
   (the member function is not already locally defined in the common superclass)

2. matchingSignaturesP(F1, F2)
   (the signatures of the member function in both subclasses match)

6.3.2 Behavior Preservation

For create_member_function: precondition 1 (the member function is not already defined locally) is satisfied by precondition 1 of this refactoring; precondition 2 (the signature is identical to that of any inherited function with the same name) is satisfied by precondition 2 of this refactoring and the *Compatible Signatures in Member Function Redefinition* program property; precondition 3 (the signatures of corresponding functions in subclasses are identical to the signature being added to the superclass) is satisfied by precondition 2 of this refactoring and by construction; precondition 4 (if there is an inherited function with the same name, either the inherited function is unreferenced on instances of C (or its subclasses), or the new function is semantically equivalent to the function it replaces) is satisfied because there are no name collisions - the new class has no local functions or instances, and its subclasses already override the function; precondition 5 (regarding references in the function body) is satisfied because the function signature has no function body.

This refactoring is constructed using a behavior preserving refactoring and, since it satisfies the preconditions of that refactorings, behavior is preserved.

6.4 Making Function Bodies Compatible

Often, in abstract classes the abstraction is represented not only by the function signatures but also by the implementations of some functions. Often these implementations are only partial; that is, the implementations include calls to other functions that are defined in the subclasses.

As in the *mapunit* example, the function bodies in the subclasses may be similar but not identical. Before the function body can be migrated to the superclass, differences need to be separated from the common code.

The approaches for detecting program differences involve string comparison, tree comparison or a combination of these techniques [12]. Program differences have been studied in regard to spelling correction [54, 109], parsing error correction [119], version storage [99] and other uses.

String comparison finds the minimum cost sequence of edit operations to convert one string into another. The standard algorithm for string comparison is described by Wagner and Fisher [120]. The user specifies the cost of insertion, replacements and deletion of particular elements. Dynamic programming is used to build a cost table; then, the table is analyzed to find the minimum cost sequence. Several refinements have been proposed [12].

String comparison algorithms have their limitations. In comparing two programs, one limitation is that often there is no way to relate changes with each other. For example, if changes are made at the beginning and end of a conditional statement, these changes will not be adjacent and string comparison approaches cannot interrelate the changes. Also, slight differences in programming style (extra parentheses, blank lines, etc.) can result in differences flagged by string comparison, even though the two programs may be syntactically the same. These and other limitations have motivated several techniques involving tree comparison.

Tree comparison algorithms detect syntactic differences between programs by building syntax trees and comparing the trees. Tree comparison is computationally expensive, and extensive research has gone into reducing the cost by first limiting the analysis to a subset of "interesting trees", and then further reducing the search by detecting and eliminating matching subtrees. Tree comparison algorithms are described in [94, 111, 115, 116, 124].

These techniques can be used in refactoring as follows: Suppose the function *abstractableFunction* is defined in classes Class1 and Class2. A string or tree comparison algorithm is applied, producing a set of insertion, replacements and deletion operations to get from *abstractableFunction* in Class1 to *abstractableFunction* in Class2. For each edit operation, define a new function in both classes:

- for each insertion, define in Class2 a new function whose body contains the inserted code; define in Class1 a new function with the same name whose body is null. Convert in Class2 the inserted code to a call to the new function; at the corresponding location within *abstractableFunction* in Class1, add a call to the new function.

- for each replacement, define in Class1 a new function whose body contains the replaced code; define in Class2 a new function with the same name whose body contains the replacing code. In Class1, convert the replaced code to a call to the new function; in Class2, convert the replacing code with a call to the new function.

- for each deletion, in Class1 define a new function whose body contains the deleted code; define in Class2 a new function with the same name whose body is null. In Class1, convert the deleted code to a call to the new function; at the corresponding location within *abstractableFunction* in Class2, add a call to the new function.

Using these algorithms, the task of comparing two functions and splitting away differences into separate functions could be automated, but there are shortcomings.

When a new function is defined, it should be given a descriptive name and contain related code. Function names that are auto-generated probably won't be descriptive. However, this is not a major problem, as a refactoring could later be applied to change the function names and make them more descriptive.

More importantly, when two functions are compared, a segment of contiguous code that appears in one function but not in the other may not belong in a new function. Perhaps the contiguous code should be divided further into several functions. Conversely, sometimes when two functions are compared, segments of code that differ between them may be preceded or followed by segments of related code that (coincidentally) are the same in both functions. In order for the new functions to represent meaningful abstractions, this "common" code might really belong together with the differing segments in those new functions. Fully automated techniques won't pick this up.

Rak [93] describes an approach that combines automated analysis with user interaction, for abstracting a function (Smalltalk method) into a superclass from its subclass implementations. A parse of each function produces a stream of tokens; the token streams are compared using

Wagner and Fisher's string comparison algorithm to determine the difference regions. On a display terminal, the functions implementations are displayed side-by-side, with the difference regions highlighted. The user can expand or split the difference regions. Then, for each difference region, a new function is added to each subclass to contain the differences. Then, in the function being abstracted, the differing code is replaced by a call to the newly created function. This process is repeated until the function implementations are identical, at which point the function can be migrated to the superclass.

Rak chose a string comparison approach because it was simpler to implement than the tree comparison approaches. However, it has shortcomings. A stream of tokens representing a difference region may cross the boundary between subtrees of the function parse, in which case that region is not always convertible to a function. This shortcoming can be overcome by expanding the difference region until it corresponds to a syntactically meaningful program part.

Beckman-Davies [12] analyzed string and tree comparison approaches (and hybrid approaches) for finding program differences. That analysis showed that no approach is optimal in all cases. Beckman-Davies further concluded that, among the approaches for finding and displaying program differences to a user, an alternative should be chosen based primarily on what seems intuitively best to that user, while minimizing computational costs and working within the limitations of the display terminal.

6.5 Making Variables Compatible

Having created the abstract superclass and determined the function signatures, it is sometimes necessary to add member variables to the abstract superclass. Although abstract classes often do not have member variables, sometimes they need to be moved there. The most common reason is that they are referenced by common code that belongs in the superclass.

As was the case with function signatures, variables defined in one subclass may be structurally similar to, but not exactly match, conceptually equivalent variables in the other subclass. For example, suppose two objects, one of each subclass, each contain a counter of references to themselves. When the counter reaches zero, the object marks itself for deletion. However, in one subclass the counter is called *refCount* while in the other subclass the counter is called *references*. Before the variable is added to the superclass, the name of one of these variables needs to be changed to match the other variable.

Structural heuristics similar to those listed in section 6.2 could detect similarities in name, access control mode and type. In cases where the attributes of the variables differ, the following refactorings that change variable attributes can be applied to make the variable attributes match, provided that their preconditions are met:

- change_variable_name (member_variable, new_variable_name)
- change_access_control_mode (member_variable, new_access_control_mode)
- change_type (member_variable, new_type).

6.6 Migrating Variables to the Superclass

This refactoring migrates a variable to the abstract superclass. The arguments to this refactoring are:

- the member variable (V1) in one subclass (C1)
- the member variable (V2) in the other subclass (C2)

After checking its preconditions, this refactoring:

1. calls the move_member_variable_to_superclass refactoring to move the variable to the superclass.

6.6.1 Preconditions

The preconditions for this refactoring are:

1. matchingAttributesP(V1, V2)
   (the attributes of the member variables match.)
2. V1.accessControlMode $\neq$ private
   (the access control mode of each variable is not private)

6.6.2 Behavior Preservation

For move_member_variable_to_superclass: precondition 1 (the variable is defined identically in all subclasses where it is defined) is satisfied by precondition 1 of this refactoring and and because (by construction, from prior steps) no other subclasses are defined; precondition 2 (the variable isn't already defined or inherited by the superclass) is satisfied by precondition 2 of this refactoring and program property four (inherited Member Variables Not Redefined), which states that a member variable inherited from a superclass is not redefined in any of its subclasses.

This refactoring is constructed using a behavior preserving refactoring and, since it satisfies the preconditions of that refactoring, behavior is preserved.

6.7 Migrating Common Code to the Abstract Superclass

In this step, common code is migrated to the superclass. Before migrating the function body to the superclass, differences between the functions are computed using one of the comparison algorithms described in section 6.4. Variables and functions referenced by the common code must be visible from the superclass before it can be moved there. The refactorings defined earlier can be applied to move variables (and add function signatures) to the superclass.

This refactoring is applied for each function body that is to be added to the superclass. The arguments to this refactoring are:

- the subclass implementations of the member function (F1 in C1, and F2 in C2) represented as a list of alternating common and differing code segments.

After checking its preconditions, this refactoring:

1. for each differing code segment, this refactoring:
   (a) calls the convert_code_segment_to_function refactoring to create a new function in each subclass. The name of the new function is automatically generated and is distinct from the name of any existing member.[5]
   (b) calls the refactoring defined in section 6.3, to add the signature of the new function to the superclass protocol
2. calls the add_function_body refactoring to add a function body to the member function signature in the superclass
3. calls the delete_member_functions refactoring to delete the member function from the subclasses.

6.7.1 Preconditions

The preconditions for this refactoring are:

1. $\exists$ member $\in$
   C1.owner.superclass.locallyDefinedMemberFunctions,
   where member.name = subclassMemberFunction1.name $\wedge$
   member.functionBody = $\emptyset$.
   (the function signature, but not the function body, is already defined in the superclass)
2. let setOfDifferingCodeSegments be the set of differing code segments for the two member functions.
   $\forall$ codeSegment $\in$ setOfDifferingCodeSegments
   convertibleToFunctionP(codeSegment).
   (each differing code segment can be converted to a legal function).
3. for each commonCodeSegment of F1 and F2,
   $\forall$ V $\in$ variablesReferencedBy(commonCodeSegment),
   $\forall$ class in (C1 $\cup$ C2 $\cup$ C1.superclass)
   class $\subset$ scopeOf(V)
   where member.name = refdVariable.name. $\vee$
   $\forall$ F $\in$ functionsCalledBy(commonCodeSegment),
   $\forall$ class in (C1 $\cup$ C2 $\cup$ C1.superclass)
   class $\subset$ scopeOf(F).
   (the scope of all variables and functions referenced by the common code includes the superclass and both subclasses).

[5]The refactoring *change_member_function_name* could later be applied to make the name more descriptive.

6.7.2 Behavior Preservation

For step 1-(a), a call to convert_code_segment_to_function: precondition 1 (each new function is a legal function) is satisfied by precondition 2 of this refactoring.

For step 1-(b), a call to the refactoring defined in section 6.3: precondition 1 (the signatures of the member function in both subclasses match) is satisfied the results of the prior step, where each new function is created in both subclasses with the same signature; precondition 2 (the member function is not locally defined in the superclass) is satisfied because by construction the new function has a unique name.

For step 2, in the call to add_function_body: precondition 1 (all variables and functions referenced in the function body are visible from the superclass) is satisfied by precondition 3 of this refactoring; precondition 2 (the member function is unreferenced locally in the superclass or the superclass is abstract) is satisfied because the superclass is abstract.

For step 3, in the call to delete_member_functions: precondition 1 (the function to be deleted is unreferenced or redundant) is satisfied because, as a result of the prior step, the function is redundant.

This refactoring is constructed using behavior preserving refactorings and, since it satisfies the preconditions of those refactorings, behavior is preserved.

6.8 Discussion

6.8.1 Reducing Interaction with the Designer

Given two concrete classes, it is possible to define a higher level refactoring that automatically creates a new common superclass that has no instances and which contains similarities detectable by structural analysis, using the following algorithm:

- create the abstract superclass, as described in section 6.1
- add function signatures to the superclass - add all identical signatures to the superclass. Going beyond exact matches, apply heuristics and refactorings described in section 6.2 to detect functions with similar signatures, refactor them to be the same, and add them to the superclass.
- move variables to the superclass - the same algorithm is applied as in the previous step, but applied to member variables, as described in section 6.6.
- for all functions whose signatures are defined in the superclass, detect common code and differing code using one of the approaches described in section 6.4. Apply the refactoring described in section 6.8 to:
  - separate differing code into new functions
  - move variables and add function signatures to the superclass as referenced by the common code, if these have not already been migrated to superclass.
  - move the common code to the superclass.

It might be that this algorithm will produce good results in practice, so that a user could just run it and then back it out if the results weren't what was intended. However it is likely that, in order to capture in the superclass an abstraction that the user can understand and later use to extend the system, some user interaction will be needed.

6.8.2 Other Issues

Given two classes, this chapter defines an approach for creating a common abstract superclass that contains a set of member function signatures, and possibly a set of member variables and the partial implementations of some functions. After refactoring, the definitions in the subclasses are streamlined, as some of the behavior that had been locally defined is now inherited. The commonalities and differences between the subclasses are made more explicit.

A related issue, outside the scope of this chapter, is whether an algorithm can be defined for selecting from a set of classes two classes that share a common abstraction. Recall from section 6.1 that the classes that shared a common abstraction were passed as arguments to the refactoring that created the superclass. Casais [30] describes a global reorganization scheme that is based on recognizing classes with common attributes. Classes are automatically brought together that share common attributes, and higher level classes are introduced to contain the common members. One shortcoming of this approach is that in choosing classes with common attributes it looks only for matching names. Also, its major motivation is to remove duplication of attributes and not to model reusable design abstractions. Thus, newly defined classes may not correspond to meaningful design abstractions and hence the design may be made less clear by the automatic reorganization.

Inferring a common abstraction from similar names and other structural similarities is not foolproof (as shown in section 6.2). Consider an extreme example, where two classes both contain the variables *dependencies, parts* and *resources*, and the functions *start, redirect, halt* and *maintain*. These classes are structurally quite similar, but if one class represents an Automobile and the other class represents a Software Project, they probably don't share a useful common abstraction. Frequently, attributes with the same name have very different meanings; conversely, classes that share a common abstraction may have attributes that are named quite differently, due to accident or history. Thus, entirely automating this process is unlikely to be successful.

The refactorings defined here assumed that the abstract superclass was created based on the commonalities between *two* classes. As noted earlier, it can be generalized to work with three or more classes. In some cases it may be desirable to create an abstract superclass with only one subclass. Confidence that a reusable abstraction has been found cannot be attained until it has been applied to more than one concrete example in the problem domain. Nonetheless, these refactorings can be applied to migrate behavior from a single class to its superclass.

Chapter 7

Refactoring To Specialize: Subclassing, and Simplifying Conditionals

7.1 Motivation

Sometimes the design of an object-oriented application framework can be improved by decomposing a large, complex class into several smaller classes. The complex class usually embodies both a general abstraction and *several different* concrete cases that are candidates for specialization. One clue for how to decompose such a complex class is to analyze the set of flags, tags and conditional statements contained in the class.

This chapter describes how to specialize a class by adding subclasses corresponding to the conditions in a conditional statement. For each condition, a subclass (representing a subtype) is created whose design abstraction implies that condition. In *C++*, the class protocol is not sufficient to describe this design abstraction because it does not make explicit the invariant conditions of a class. To represent this design information, the class protocol is supplemented with a *class invariant*.

Specializing a class and simplifying conditionals involves several steps:

1. choosing a conditional whose conditions suggest subclasses
2. for each condition in the conditional, creating a subclass with a class invariant that matches the condition
3. copying the body of the conditional to each subclass, and in each subclass simplifying the conditional based on the invariant that is true for the subclass
4. specializing some (or all) expressions that create instances of the superclass. Specializing an expression involves replacing an expression that creates an instance of the superclass with an expression that creates an instance of one of the newly created subclass. This can be done safely only if all instances created by the expression are known to satisfy the invariant defined on a subclass.

Choosing the 'correct' conditional upon which to partition a class (step one) depends on the abstraction that the designer intends. A tool could assist the designer in making this choice by displaying the conditionals that appear in member functions of the class, highlighting those conditionals that appear most frequently.

This chapter defines refactorings for steps two through four. These steps are defined here as separate refactorings, mostly for clarity but also because they could be invoked separately. Specializing instances can sometimes be done automatically, but in some cases an instance of the superclass might satisfy the invariants of several subclasses. Choosing among these subclasses may be done incorrectly without some input from the user.

The techniques defined in this chapter rely heavily on data flow analysis to check that class invariants hold on instances of a class. Class invariants and data flow analysis are described, respectively, in sections 7.2 and 7.3. Sections 7.4 through 7.9 describe how to use data flow analysis to handle several important issues regarding class invariants, and define refactorings. Section 7.10 discusses possible extensions to the subclassing algorithms defined here; other uses of class invariants in the object oriented literature are also discussed.

This thesis specifically addresses how class invariants can be used for simplifying conditionals in member functions of newly created subclasses. Class invariants could be applied more generally in refactoring, which is an area for future research.

7.2 Class Invariants

An *invariant* is something that remains constant (i.e. does not change). A *class invariant* is a predicate whose only free variables are member variables of the class. A class invariant expresses general consistency constraints that apply to every instance of a class.

In general, determining if an arbitrary predicate is a class invariant is undecidable. For the purposes of this thesis, an extremely narrow set of class invariants are considered [1] which are of the form:

```
<memberVariable> == <value>
```

where:

```
memberVariable:  a member variable of the class
value:           an integer, or the value of an enumerated type.
```

For the class invariant to be correct:

- the *create* procedure for the class (a constructor function in C++), when applied, must yield a state satisfying the class invariant
- the class invariant must hold throughout the remainder of the lifetime of each instance of the class.

The algorithm for subclassing and simplifying conditionals, outlined in section 7.1, involves class invariants at several steps. Two important problems arise regarding class invariants:

1. How can it be determined whether a predicate is a class invariant for a class?

[1] There are decision procedures for less restrictive sets of class invariants.

2. Given a correct class invariant, how can it be used, to:
   (a) simplify a conditional statement?
   (b) specialize an expression that creates an instance of the superclass?

None of these issues is decidable, in general. However, they are decidable in some cases, as discussed later in this chapter.

7.2.1 Example

To illustrate how a class invariant could be used to simplify a conditional, recall the example shown in Figure 7.1 and first presented in chapter 3. In the *Typed Smalltalk* (TS) compiler, instances of the FlowNode class, representing a basic block, were containers of a sequence of straight line code (assignments) ending with a statement that alters the flow of control. The ending statement in the sequence could be a return statement, a conditional jump statement or an unconditional jump statement. In the early designs, it had not been clear that the type of the ending statement was an important discriminator among FlowNodes. However, as the design matured, the implementation of FlowNode was manually refactored: FlowNode became an abstract superclass and the concrete subclasses UncondJumpFlowNode, CondJumpFlowNode and ReturnFlowNode were added.

The value of the opcode in the ending statement is represented as the member variable *final_statement* in the FlowNode class. Suppose that the value of *final_statement* is assigned in the *create* procedure of the FlowNode class (and its subclasses, and that there are no assignments to change the value of that member variable. Then, in the FlowNode example above, the predicate assigned as the class invariant of the class CondJumpFlowNode would be:

```
final_statement == conditional_jump
```

Note that this predicate is specified in terms of a fixed value of the member variable. This invariant could be used to simplify the conditional expression in the *optimize* function, removing the conditional test and all other code except a code segment (S1 in CondJumpFlowNode, S2 in UncondJumpFlowNode or S3 in the ReturnFlowNode class).

In general, many conditions in a conditional statement are of the form:

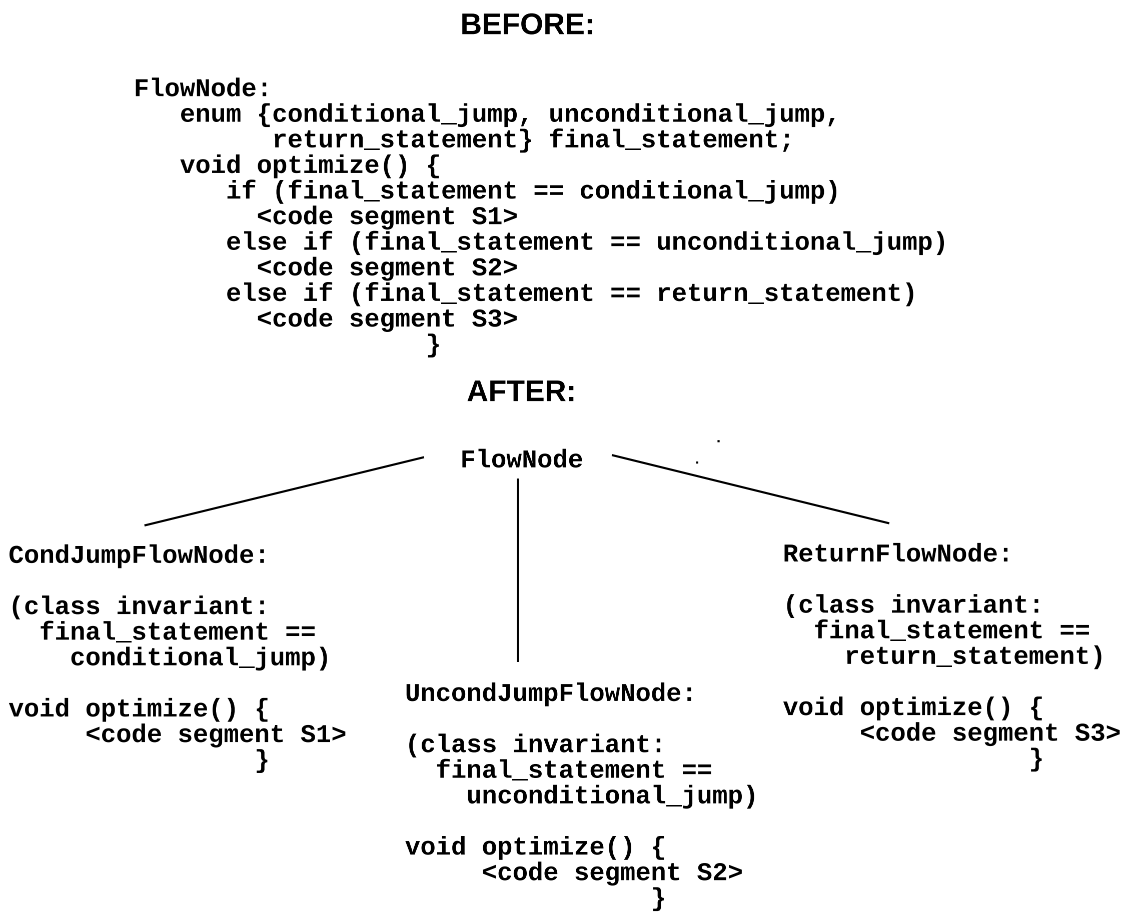
```
<memberVariable>==<value>
```

where *value* is a constant. The value of *memberVariable* is often set by an assignment whose right-hand-side is either a constant or a parameter to the function containing the assignment. In the latter case, the argument in calls to the function is likely to be a constant. These cases are likely to make up the vast majority of conditional simplifications,[2] and are easy to check.

7.2.2 Discussion

Data flow analysis techniques are both safe and powerful enough to handle some cases involving predicates being asserted as class invariants. The following section provides background on data flow analysis.

[2] Determining what percentage of conditionals are covered by these cases would require an exhaustive study of programs, beyond the scope of this research. Software engineering studies of refactoring large programs, written in several practical (albeit complicated) languages with several different programming styles, is an area for future research noted in chapter 11.


BEFORE:

```
FlowNode:
   enum {conditional_jump, unconditional_jump,
         return_statement} final_statement;
   void optimize() {
      if (final_statement == conditional_jump)
        <code segment S1>
      else if (final_statement == unconditional_jump)
        <code segment S2>
      else if (final_statement == return_statement)
        <code segment S3>
                   }
```

AFTER:

```
FlowNode
```

```
CondJumpFlowNode:

(class invariant:
  final_statement ==
    conditional_jump)

void optimize() {
     <code segment S1>
                }
```

```
UncondJumpFlowNode:

(class invariant:
  final_statement ==
    unconditional_jump)

void optimize() {
     <code segment S2>
                }
```

```
ReturnFlowNode:

(class invariant:
  final_statement ==
    return_statement)

void optimize() {
     <code segment S3>
                }
```

Figure 7.1: Simplifying Member Function *optimize*

7.3 Background: Data Flow Analysis

A refactoring that simplifies a conditional statement can be thought of as a kind of code optimization. Some techniques that help in optimizing code for compilers can also be applied in refactoring.[3] Optimizing a program involves two tasks: analyzing the program and modifying it. In general, the analysis step is the most difficult.

Local optimizations are changes that are applied to sequential code segments (*basic blocks*). They are the easiest optimizations to implement and, until recently, compilers rarely did more than local optimization. Examples of local optimizations are evaluating (within a basic block) constant expressions in advance and eliminating some useless instructions.

[3] Not all techniques apply in refactoring, since the motivation for refactoring is different from the motivation for optimizing code in compilers. In compilers the optimizations are made to increase the speed or reduce the memory size of an implementation; the resultant code is not meant to be human readable. Some compiler optimizations restructure the code in a way that makes it more difficult to understand by a human designer. By contrast, the motivation for refactoring is to produce a design than can be more clearly understood by a designer.

Global optimizations are more complex than local optimizations, because they have to deal with the flow of control across basic blocks. Refactorings that simplify conditionals can be thought of as a kind of global optimization.

To optimize a program or a procedure within a program, it is usually necessary to determine the sets of variables read and updated by that procedure. Unless these sets can be determined, worst case assumptions must be made. For example, if the procedure includes a call to another procedure, in the absence of information about the called procedure it must be assumed that all variables visible to the called procedure will be read and updated. This assumption, while safe, prevents many optimizations. One can expect to do better if the effects of a call are more carefully analyzed. The analysis of the effects of a call is generally termed *interprocedural data flow analysis.* [1, 44, 106]

7.3.1 Interprocedural Data Flow Analysis

A procedure or subprogram can be represented as a *data flow graph* of basic blocks, where directed edges connecting the basic blocks represent the flow of control within the procedure. The calling relationships among procedures in a program can be represented using a *call graph*, where each node represents a procedure and directed arcs represent calls between procedures.

The flow of control for the entire program can be represented using a *data flow graph* for each procedure and a *call graph* that interrelates the procedures. It is not always possible to determine statically what sequence of basic blocks a program will execute; therefore, it is usually assumed that all paths in each data flow graph are possible.

Two useful sets to associate with each procedure call are *Def* and *Use*. A *Def* set is the set of variables that may be defined (assigned) during a call, and the *Use* set is the set of all variables or named constants that may be used (read) in the call. The *Def* and *Use* sets for a procedure can be computed from its data flow graph. The sets defined for a called procedure can help determine what optimizations are possible in the procedures that call it.

Some data flow problems can be described as *any path* problems, where the analysis is concerned with whether a property holds on one or more possible paths through a program. Other data flow problems are *all path* problems, where it is necessary to determine whether a property holds over all paths.

Some data flow problems are solved using *forward flow* analysis, following the direction of flow in a program. Consider, for example, computing for a variable reference the set of assignments to that variable (the *reaching definitions*) that may precede it without an intervening reassignment. This analysis can be done by finding all assignments to the variable, and moving forward through the flow graph to see if the variable reference in question can be reached.

Other data flow problems are solved using *backward flow* analysis, where the direction of analysis is opposite the flow of control of the program. For example, to construct the set of references to a variable that may follow a particular assignment (often called a *definition-use chain* or *du-chain*), analysis can be done by finding all references to that variable, and moving backward through the flow of control to determine if the assignment in question can be reached.

Many data flow problems can be solved by associating with each basic block or procedure a series of "sets":

- the *Gen* and *Killed* sets, which are (respectively) the sets of definitions that are generated by (or invalidated by) a block. For example, if the block contained a single assignment statement of the form:

```
X = 2;
```

then the *Gen* set (for variable X) would the value 2, and the *Killed* set (for variable X) would be all other values of X that might have reached the block.

- the *In* and *Out* sets, which are (respectively) the sets of definitions that reach (or exit) a block. These sets are constructed from *Gen* and *Killed* sets of blocks and the program flow relationships between basic blocks.

Solving a particular problem involves defining and generating the *Gen* and *Killed* sets for each basic block, and characterizing the problem as either an any path or all path problem, and as either a forward flow or backward flow problem. For example, the reaching definitions problem is an any path, forward flow problem. The *Gen* set for a block is defined to contain the definitions that appear in the block and reach the end of the block. The *Killed* set for a block is defined to contain the definitions that appear somewhere in the block but do not reach the end of the block. By contrast, constructing the *du-chain* involves any path, backward flow analysis. The *Gen* set for a block is defined to contain the set of statements that use a variable before it is defined in that block. The *Killed* set for a block is defined to contain the set of statements that use the variables after it is (re)defined in the block.

After the *Gen* and *Killed* sets have been determined for each basic block, the *In* and *Out* sets can be determined using flow graphs and generic data flow equations such as described in [44].

Two common approaches for solving data flow problems are *iterative* and *structured* approaches [44]. Iterative approaches [56, 118] repeatedly traverse the flow graphs until there are no more changes. Structured solutions (e.g., [7, 67, 107]) are more complex, but have much better worse case performance.

7.4 Checking Whether a Predicate is a Class Invariant

In general, the problem of determining whether an arbitrary predicate is a class invariant is undecidable. The approach taken here is to restrict the language of class invariants, and apply conservative analysis techniques. Fortunately, the language of class invariants (defined earlier, in section 7.2) is expressive enough, and data flow analysis techniques powerful enough, to handle some common cases. Data flow techniques are conservative; if a class invariant is shown to be correct using data flow techniques then it is correct, but there will be cases where a predicate is a class invariant but it cannot be verified using data flow analysis.[4]

One feature of these predicates that makes them difficult to validate is that each clause asserts a value (or range of values) of a member variable; a member variable can change its value over time. For example, the assertion:

```
Y==3
```

may be true when an object is first initialized, but if an operation is invoked that assigns another value to member variable *Y*, then the assertion no longer holds. Therefore, to determine that a predicate holds, it must be shown that:

[4]There are program analysis and verification techniques more powerful than data flow, but the analysis is more expensive and is still not always decidable.

- the *create* procedure returns a state implied by the predicate
- the predicate implies the range of values that each member variable (in the predicate) might hold after the object is created.

The first condition can be easily determined if, for all member variables referenced in the predicate, either the *create* procedure assigns a constant value to each variable, or each variable is assigned the value of an argument to the create procedure and the procedure is always called with constant arguments.

Determining the range of values that a variable might hold after it is created requires finding all places where it may be assigned a value, and what values may be assigned there. Interprocedural all paths data flow analysis is a "safe" technique that can locate all assignments that might be made to a variable.

Data flow analysis can determine where assignments to a variable are made and what expressions are assigned to the variable. Determining whether a predicate is violated requires knowing the *value* of the expressions assigned to the variable. In many common cases the assigned value is a constant, in which case the value is known directly, or the variable is assigned the value of another variable, in which case data flow analysis can be used to attempt to determine the set of reaching values of that variable. Data flow analysis is less likely to work with more complicated expressions.

Several factors influence the scope of the analysis. The access control mode of a variable (private, protected or public) determines the range of functions that may reference the variable. Also, if the variable is passed by reference in a function call, the function and the transitive closure of the set of functions which might receive the argument by reference must be checked.

Data flow analysis is less likely to be able to detect when a predicate holds if the variable being analyzed is part of a variant field, or pointers to the variable are passed and manipulated using pointer arithmetic and other (dangerous) operations. Such features make it difficult or impossible to determine where a variable is assigned. Fortunately, few object oriented languages allow pointer arithmetic and similar operations. C++ is an exception, since it is a superset of C. However, subclassing can eliminate the need for variant structures in C++, and C++ encourages a style that involves fewer pointer manipulations than does C.

7.5 Refactoring: Create Subclasses and Assign Invariants

The first important step in the refactoring process is to create the new subclasses, and assign the class invariants. This refactoring creates subclasses that correspond to the enumerated values of a member variable.

The arguments to this refactoring are:

- the class being specialized (C)
- the member variable (enumeratedMemberVar) was enumerated values are used to create subclasses.

After checking its preconditions, this refactoring (for each enumerated value):

1. calls the create_empty_class refactoring to create a subclass, whose name corresponds to the enumerated value

2. calls the create_member_function refactoring to add a constructor (*ie* create) function to the subclass. The constructor function establishes initial conditions implied by the class invariant. For example, if the class invariant is:

   ```
   X==3,
   ```

   the constructor will assign the value *3* to member variable *X*.

3. assigns a predicate to the class as the class invariant.

7.5.1 Preconditions

The preconditions for this refactoring are:

- $\forall$ C $\in$ Program.classes,
      $\forall$ enumVal $\in$ enumeratedMemberVar.type,
          C.name $\neq$ enumVal.name.
  (none of the names of the enumerated values clash with the names of existing classes).

- $\forall$ F $\in$ (C.locallyDefinedMemberFunctions $\cup$ inheritedMemberFunctions(C)),
      $\forall$ enumVal $\in$ enumeratedMemberVar.type,
          F.name $\neq$ enumVal.name.
  (none of the names of the enumerated values clash with the names of member functions of the superclass).

7.5.2 Behavior Preservation

For create_empty_class: precondition 1 (the name doesn't clash with an already existing class) is satisfied by precondition 1 of this refactoring.

For create_member_function: preconditions 1 and 2 (the name of the constructor function doesn't clash with an already locally defined member function, and has a compatible signature with any inherited function with the same name) are satisfied by precondition 2 of this refactoring; precondition 3 (regarding subclasses of each newly defined class) is satisfied because the newly created classes by construction have no subclasses; precondition 4 (if there is an inherited function with the same name, either the inherited function is unreferenced on instances of C (or its subclasses), or the new function is semantically equivalent to the function it replaces) is satisfied by precondition 2 of this refactoring; precondition 5 (the function will compile) is satisfied because the member variables are visible to the constructor, and the assignment is type-correct.

The third step (attaching a predicate as the class invariant) does not change the behavior of the program because it is design information and does not change the program source code.[5]

This refactoring is constructed using behavior preserving refactorings and, since it satisfies the preconditions of those refactorings, behavior is preserved.

[5]However, by precondition 2 and the results of the prior step, at this step the class does not violate the predicate attached as the class invariant.

7.6 Using A Class Invariant To Simplify A Conditional

Given a predicate known to be a correct class invariant, this section describes how to simplify conditional statements in the functions defined for the class. The algorithm is:

1. for each condition in a conditional statement, reduce to true or false if possible based on the class invariant;
2. eliminate dead code and perform other simplifications.

The first step is a special case of theorem proving and, in general, is undecidable. However, since class invariants are of the form:

```
<memberVariable> == <value>
```

where the value is an integer or the value of an enumerated type, *constant propagation* and *copy propagation* [44] can be used to reduce some conditions to true or false.

Constant propagation and copy propagation are applied in compilers to detect and remove extraneous variables, and improve execution speed by replacing variable references with their values. Constant propagation means that, for example, if the assignment:

```
X=3
```

is followed by a statement that references the variable *X*, and it is known that the value of *X* is not reassigned by intermediate statements, then in the latter statement the reference to *X* can be replaced by the value *3*. Copy propagation means that, for example, if the assignment:

```
Y=X
```

is followed by a statement that references *Y*, and it is known that neither *X* nor *Y* are reassigned by intermediate statements, then in the latter statement the reference to *Y* can be replaced by a reference to *X*. There are many algorithms for simplifying expressions by performing constant propagation and copy propagation [44].

These techniques can be applied for simplifying conditional statements. For a variable referenced in a condition, an any path, forward flow analysis can be used to compute the set of *reaching definitions*, that is, the set of definitions to the variable that might reach the reference. If only one assignment to a variable can "reach" a reference to that variable, then that variable reference can be replaced by the assigned value.

Consider how the conditions in the above FlowNode example can be automatically simplified. In the class CondJumpFlowNode, constant propagation based on the class invariant replaces references to the variable *final_statement* with the value *conditional_jump*, resulting in the following code:

```
void optimize() {
   if (conditional_jump == conditional_jump)
     <code segment S1>
   else if (conditional_jump == unconditional_jump)
     <code segment S2>
   else if (conditional_jump == return_statement)
     <code segment S3>
                 }
```

Eliminating dead code and performing other simplifications is the second step in the general algorithm. In the above example, for the class CondJumpFlowNode the first condition is trivially true and the conditional can be reduced to code segment S1. Similarly, the conditional statement can be reduced to a code segment in the other two subclasses. This step would be more complicated if for example the conditional tests had side effects.

More powerful simplifications can be performed by applying constant and copy propagation using class invariants and other program properties determinable using data flow analysis. These techniques could be applied to simplify some conditionals that test the value returned from a function call, if the values returned from the function could be enumerated. These techniques can also be applied to simplify expressions other than conditional statements.

7.7 Refactoring: Migrate & Simplify Conditionals

In this step, the conditional statements to be simplified are separated into new functions, which are copied into the subclasses and simplified.

This refactoring simplifies *a* conditional statement. It separates the conditional into a separate function, copies it into the subclasses and, in each subclass, simplifies the conditional based on the class invariant. The argument to this refactoring is:

- the conditional statement

After checking its precondition (which is the same as the precondition for the primitive refactoring convert_code_segment_to_function), this refactoring:

1. calls the convert_code_segment_to_function refactoring to separate the conditional into a separate function, assigning it a unique name

2. constructs the sets of all member functions and variables referenced by the conditional. For each member where member.accessControlMode = private, calls change_access_control_mode to make it protected.

3. calls the create_member_function refactoring to copy the new function to each of the new subclasses

4. if firstConditionImpliedByInvariant(conditional statement, class invariant of subclass) can be determined:

   (a) simplifies the conditional statement.

   (b) eliminates all branches that test false (without side-effects) against the class invariant.

   (c) converts any branch that tests false (but has side-effects) to a statement corresponding to the side-effect.[6]

[6] If all conditions test false (without side-effects), the entire conditional statement can be eliminated. Detecting side effects is in general difficult.

7.7.1 Preconditions

The preconditions for this refactoring are:

- convertibleToFunctionP(conditionalStatement)
  (the conditional statement is convertible to a legal function)

7.7.2 Behavior Preservation

For convert_code_segment_to_function: precondition 1 (the conditional is a legal function) is satisfied by precondition 1 of this refactoring.

For change_access_control_mode: preconditions 1 and 3 are satisfied, since they only apply if the access control mode is not private; precondition 2 (variables referenced in the common code are not private variables that are redefined in subclasses) is satisfied because by construction, before this step the variable or function is not locally defined in the subclasses.

For create_member_function: preconditions 1-4 (regarding name clashes and equivalence with functions inherited from the superclass) are satisfied because in the prior step it was added to the superclass and given a unique name, while in this step it is copied down; precondition 5 (regarding all variable and function references in the conditional are visible from the subclasses) is satisfied by the results of a prior step.

For step 4: program property seven (semantically equivalent references and operations) is preserved by construction - side effects of conditional tests are retained; only dead code is removed. Other program properties are trivially preserved.

This refactoring is constructed using three behavior preserving refactorings, for which it satisfies the preconditions, and a fourth operation which by construction is behavior preserving. Therefore, behavior is preserved.

7.8 Using A Class Invariant To Specialize Expressions That Create Instances

After the new subclasses have been defined, the design of a program can sometimes be improved by specializing the expressions that create instances of the superclass; that is, replace an expression that creates an instance of the superclass with an expression that creates an instance of a subclass.

To extend the above example involving FlowNodes, consider a class ProgramBuilder that builds new FlowNodes. In one of its functions, the statement to create a FlowNode is:

```
newNode = new FlowNode (<list of accumulated instructions>,
                        unconditional_jump)
```

The constructor for the class Flow Node begins with an assignment of the second argument to the member variable *final_statement.* Thus, for this assignment statement, the second argument passed to the constructor function corresponds to the class invariant of UncondJumpFlowNode. Suppose that member variable *final_statement* is never reassigned. In the above assignment, the expression creating an instance of the superclass FlowNode can be replaced with an expression creating an instance of the subclass UncondJumpFlowNode; that is:

```
newNode = new UncondJumpFlowNode (<list of accumulated instructions>)
```

In general, this operation cannot be done safely unless it can be determined that no instance created by the expression violates the class invariant of the subclass. Determining this involves finding where the instances created by this expression are initially assigned, recording the variable initially assigned the instances and constructing a definition-use chain for that variable, to find all places where the variable may be used. If the variable is aliased (e.g., it is passed by reference in a function call), a definition-use chain for all variables to which it may be aliased must also be constructed. In order to determine that the instance can be specialized, it must be proven that none of the operations on these variables violate the class invariant defined on a subclass.

7.9 Refactoring: Specialize Expressions That Create Instances

This refactoring specializes an expression that create(s) instances of the superclass with an expression that creates instance(s) of one of its subclasses, as described in section 7.8.[7]

The arguments to this refactoring are:

- the expression that creates an instance of the superclass (C)
- the replacing expression that creates an instance of the new subclass

After checking its preconditions, this refactoring:

1. replaces the expression with an equivalent expression that creates instance(s) of the subclass.

7.9.1 Preconditions

The preconditions for this refactoring are:

1. satisfiesClassInvariantP (expression, subclass)
   (instances created by the expression do not violate the subclass invariant, as discussed in section 7.8).

2. $\forall$ V $\in$ (inheritedMemberVariables(C) $\cup$ C.locallyDefinedMemberVariables),
        $\exists$ V2 $\in$ (inheritedMemberVariables(subclass) $\cup$
            subclass.locallyDefinedMemberVariables)
        $\wedge$ matchingAttributesP(V, V2).
   (for each variable in the superclass there is a matching variable in the subclass).

3. $\forall$ F $\in$ (inheritedMemberFunctions(C) $\cup$ C.locallyDefinedMemberFunctions),
        $\exists$ F2 $\in$ (inheritedMemberFunctions(subclass) $\cup$
            subclass.locallyDefinedMemberFunctions)

[7]In general, determining whether an expression can be specialized is undecidable. In a program, there may be some expressions that cannot be specialized automatically, either because determining whether the precondition is true is undecidable, or because the instance violates the invariants of all of the subclasses.

$\wedge$ matchingSignatureP(F,F2)

$\wedge$ given the subclass invariant, F and F2 are semantically equivalent.

(for each function in the superclass, there is a function in the subclass with a matching signature and which is semantically equivalent, given the subclass invariant).

4. if the expression is a declaration of a (non-pointer) variable[8]:

   (a) there are no assignments *to* the variable

   (b) assignments *of* the variable remain type safe.[9]

7.9.2 Behavior Preservation

For step one, program property six (type-safe assignments) is preserved by precondition 4 (if the expression is a declaration of a (non-pointer) variable); otherwise, program property six is preserved because the variable assigned the result of the expression must (by definition) accept an instance of the superclass and hence will accept an instance of one if its subclasses. Also for step one, program property seven (semantically equivalent references and operations) is preserved by preconditions 1 - 3 of this refactoring; other program properties trivially preserved. Thus, behavior is preserved.

7.10 Discussion

7.10.1 Possible Extensions In Subclassing

Sometimes, additional improvements can be made to the resulting implementation. If there are no instances of the superclass remaining, unreferenced functions can then be deleted from the (now, abstract) superclass. If there are any member variables defined in the superclass that are only referenced in a subset of the subclasses, then they can be migrated down to the subclasses.

While the refactorings in this chapter specifically consider specializations based on the conditions of a conditional statement, class invariants as described here can more generally be applied for other specializations.

There are cases where the state of an instance may change in a way that violates the invariants defined on the subclasses. Consider the TS Compiler example, discussed earlier. Suppose that the final statement in a FlowNode could dynamically change, for example from a conditional jump to an unconditional jump, as part of a compiler optimization. If that instance were first specialized to be an instance of the CondJumpFlowNode class, then when the final statement of the flow node is changed that instance should be changed to or replaced by an equivalent instance of the UncondJumpFlowNode class.

The refactoring in section 7.9 disallows such an instance to be specialized, since it violates the first precondition. There are other alternatives. One is to replace an instance of the current subclass with an instance of a new subclass, when the state of an instance changes in a way that violates the invariant of its class. This approach requires finding all references to the old instance and changing them, along with copying any state information from the old instance needed by the new instance. This approach worked in the *TypedSmalltalk* example but, in

[8] See discussion of the convert_instance_variable_to_pointer refactoring.

[9] See precondition two of the change_type refactoring.

general, could not be applied unless all parts a program that might point to the old instance were known. Another alternative is to refactor, into a new component class, the behavior that differentiates the cases for which subclasses should be created. Then, specialize the component class and substitute components when the condition changes. In this case, the original class is not replaced when the condition changes, but its component is replaced by a component of another subclass. Refactorings involving components and the aggregate objects that contain them are defined in the next chapter.

7.10.2 Other Uses of Class Invariants

The term *class invariant* is used in several other ways in the object-oriented literature. In refactoring, the term *class invariant* is used in a manner similar to Meyer's [79] use of the term.

Meyer describes a class invariant as a list of assertions that apply to every instance of a class.[10] His *invariant rule* states that, for a class invariant to be correct:

- invoking the *create* procedure for the class yields a state satisfying the class invariant
- every exported routine of the class, when invoked in a state satisfying the class invariant, will yield a state satisfying the class invariant.[11]

In this rule:

- every class is considered to have a *create* procedure
- the state of the object is defined by the values of its attribute fields
- the class invariant may only involve the state of an object.

This definition is similar to the above definition used in refactoring, but there is an important difference. Meyer uses class invariants to ensure correctness at the interfaces between classes. For these purposes, the class invariant need only hold at the points where an instance of the class is called by another object; the class invariant need not hold at other points during the execution of an exported routine, nor need it hold when the class calls an internal (private) routine. By contrast, refactoring uses a class invariant to simplify the internals of the class; for these purposes the invariant must also remain true *as viewed from inside the class.*

Another use of the term *class invariant* is made by Casais in [31]. Casais defines class invariants as integrity constraints that must be maintained across changes to the schema (system structure).[12] These constraints include, for example, the *full inheritance invariant*, which states that a class inherits all attributes from its ancestors (except those that it explicitly redefines), and the *type compatibility invariant*, which states that the type of a variable redefined in a subclass must be consistent with its domain as specified in the superclass. Invariants such as these are important for understanding the semantics of the language being used, but they are distinct from the constraints defined on specific classes.

[10] He distinguishes these from preconditions and postconditions defined on routines.

[11] His invariant rule also requires that, when these procedures are applied, the arguments and state satisfy the preconditions for the procedures.

[12] Casais [31] compares the GemStone® [92], Orion [9], O_2 and OTGen object-oriented database systems with respect to class invariants. GemStone is a registered trademark of Servio Logic.

In summary, in refactoring the term *class invariant* is used in a manner similar to Meyer's [79] use of the term, although in refactoring the invariant applies not only where the class calls other classes, but also for states internal to the class.

Chapter 8

Refactoring To Capture Aggregations & Components

8.1 Motivation

Inheritance is a powerful technique, but in modeling the relationships among classes it is sometimes overused and incorrectly used. *Aggregations* are another tool for modeling these relationships.

To summarize from chapter 3, an aggregation is a special association between objects, representing a part-whole relationship. An aggregate object, sometimes called a composite object or a container, contains one of more parts, called components. An aggregate class is a class containing one or more component members; a component class is a class whose instances are components of another class.

Components are stored as member variables in the aggregate object; however, as described in chapter three, not all member variables of an aggregate class store component objects. For example, member variables in an Automobile class might represent its color, age, or number of passengers; these member variables represent attributes of an automobile but are not components of it, in the sense that an engine or left front tire is a component of an automobile.

An aggregate object can be treated as a unit for many operations. During design, recognizing that an association between objects is an aggregation can help determine how to partition behavior and responsibilities among the classes. Sometimes it is not obvious that the relationship between objects is an aggregation until the code is reused; as a program is reused the relationships between classes become more obvious. In these cases, correctly partitioning the behavior may require moving members between aggregate classes and the classes of their components. These changes may not only improve the design of the aggregate class, but also make the component classes more reusable.

An example of where such a move might be made is when the Automobile class contains a component member variable *autoEngine*, which holds an instance of the class Engine. The class Engine contains the member variable *milesPerGallon* which, for design reasons, really belongs in the (aggregate) class Automobile. The refactoring described in section 8.7 defines how to automate such a design change.

This chapter defines several refactorings involving aggregations:

- moving members from an aggregate class to the class of one of its components

- moving members from a component class to the aggregate classes that contain component members that are instances of the component class
- converting a relationship, modeled using inheritance, into an aggregation.

Each of these refactorings requires that the components of a class be determinable. For example, moving a member safely from an aggregate to a component class (or vice-versa), requires that there must be a one-for-one mapping between the member being removed from the old class and the member being added to its new class (or set of classes). *For the purposes of refactoring, an important characteristic of components is that a component object can only be assigned to one aggregate object at a time.*[1] Put another way, objects assigned to one component member variable must not simultaneously be assigned to another component member variable. For example, in an Automobile object the same Tire object cannot be assigned as both the *leftFrontTire* and *rightFrontTire*; similarly, it cannot be assigned to two Automobiles at the same time.

The following section defines some algorithms for checking that a member variable qualifies as a component member variable. The subsequent sections define the refactorings. The chapter closes with a discussion of related issues.

8.2 Checking If A Variable Qualifies As A Component

There are several ways to determine whether a variable qualifies as a component member variable; that is, that every object assigned to it is not also assigned to another component variable. One common case that is trivial to check is when the variable is only directly assigned the result of an *object creation expression* (i.e. by a call to the C++ *new* function) and its value is not reassigned to another variable.

For example, the *Choices* Virtual Memory system [104] has two kinds of objects, MemoryObjects and MemoryObjectCaches. A MemoryObjectCache is a component of a MemoryObject. A MemoryObject does not always have a MemoryObjectCache, but once it gets one it never changes. The code to initialize the member variable *cache* looks something like this:

```
if (cache == 0) {
     cache = new MemoryObjectCache(this) }
```

Provided that the value of member variable *cache* is not assigned to another variable, it qualifies as a component for the purposes of refactoring.

The algorithms defined below handle the more general case where the variable may not be private to its class, or it may be assigned an object created in another (server) class. The section is structured in four parts:

1. it describes a conservative data flow algorithm that assumes that the program contains a single instance of the aggregate class;
2. it extends the initial algorithm to better handle cases where component objects are created by classes other than the aggregate class;

[1]This definition is consistent with the meaning of *exclusive* composite references in object-oriented databases, as defined by Kim [66].

3. it further extends the algorithm to handle the case where a program contains multiple instances of the aggregate class;
4. finally, it discusses additional extensions that could make the algorithm more powerful.

The assumption in parts one and two that the program contains a single instance of the aggregate class is very restrictive, but these parts illustrate important points. Parts three and four describe less restrictive extensions, that build upon the analysis in the earlier parts.

8.2.1 Assumptions and Terminology

To simplify the presentation, the following assumptions are made about the program:

- all objects are created dynamically by an *object creation expression* (i.e. a point in the program where the C++ *new* function is called), rather than being statically assigned to a variable at compile time.
- each return statement of a function is of the form: *return* V_x, where V_x is a variable.
- the set of aliasing relationships are known; that is it is known which variables are aliased to other variables. C++ reference variables (where a variable is passed by reference in a function call) are an example of aliasing. Aliasing relationships are more difficult to determine when pointers are passed.

All variables (global, member, argument and local) are uniquely identified. However, all references to the same member variable, even if the references are to different instances of the same class, will map to the same identifier.

Variables have their values set by assignment statements and (for argument variables) by function calls. A variable (V_x) may be assigned:

1. the value of another variable (V_y).
2. a constant (C_i)
3. the return value of a function ($F_j()$)
4. an object returned by an object_creation_expression (O_z)

Associated with each variable V_x is an (initially empty) set D_{V_x} of object creation expressions. When analysis is complete, D_{V_x} will contain the set of all object creation expressions whose return values (newly created objects) may be assigned to V_x.

R_{F_j} denotes the set of variables that appear in return statements of function $F_j()$.

$D_{R_{F_j}}$ denotes the set of object creation expressions whose objects may be returned by function $F_j()$; that is, $D_{R_{F_j}} = (\bigcup_{V_y \in R_{F_j}} D_{V_y})$. For example, if function $F_j()$ contained two return statements: *return* V_x and *return* V_y, then $D_{R_{F_j}} = D_{V_x} \cup D_{V_y}$.

8.2.2 Basic Algorithm: Single Instance of Aggregate Class

Associated with each assignment (assignment statement and function call) is a set of object creation expressions that are added to D_{V_x}, where V_x is the variable on the left hand side of the assignment. The *gen* sets for each assignment statement S are as follows:[2] [3]

Assignment (S)	gen[S] (added to D_{V_x})
$V_x = V_y$	D_{V_y}
$V_x = C_i$	$\emptyset$
$V_x = F_j()$	$D_{R_{F_j}}$
$V_x = O_z$	$\{O_z\}$

After D_{V_x} has been computed (using data flow techniques) for all variables in the program, the sets of object creation expressions for aliased variables must be merged; that is, if V_x is aliased as V_y, then: $D_{V_x} = D_{V_y} = (D_{V_x} \cup D_{V_y}.)$

Assuming that the program contains only a single instance of the aggregate class, at this point it can be determined whether a variable V_c can be designated as a component variable. It can be designated as such if all values assigned to it are not assigned to other component variables. This is certainly true if all expressions that create objects reaching V_c do not also create objects reaching a variable already designated as a component variable, that is:

```
ComponentIfSingleAggregateInstanceP(variable $V_c$) $\equiv$
    $\forall\ V_y \in$ componentVariables(Program),
        $V_y \neq V_c \Rightarrow$
            $D_{V_y} \cap D_{V_c} = \emptyset$.
```

8.2.3 An Improvement to the Basic Algorithm

One way in which the above approach is conservative is that it does not distinguish among objects created by the same object creation expression. Suppose there is a function $F_j()$ (a tire distributor) that returns a new tire with each call to it. One of the member functions of the Automobile class contains the code segment: *leftFrontTire = F_j(); rightFrontTire = F_j();*. With the above algorithm, $D_{R_{F_j}}$ would be added to both $D_{leftFrontTire}$ and $D_{rightFrontTire}$; since $D_{leftFrontTire}$ and $D_{rightFrontTire}$ would contain common object creation expressions, the above analysis would conclude that they cannot both be designated as a component member variable. However, since the variables are assigned different tires, they may qualify as component variables.

To handle this, the data flow equations are changed to:

[2] The *Killed* sets are null

[3] In C++, the variable *this* referenced in a member function points to the object for which the function is called [41]. For an assignment of the form $V_x = V_y$, where V_y is the variable *this*, add to D_{V_x} the set of all object creation expressions that create instances of classes that define or inherit the member function containing the assignment.

Assignment (S)	gen[S] (added to D_{V_x})
$V_x = V_y$	D_{V_y}
$V_x = C_i$	$\emptyset$
$V_x = F_j()$	$D'_{R_{F_j}}$
$V_x = O_z$	$\{O_z\}$

This is the same as the above set, except that for a function call the set $D'_{R_{F_j}}$ (rather than $D_{R_{F_j}}$) is added to D_{V_x}.

$D'_{R_{F_j}}$ is defined as follows:

- if flow analysis showed that $F_j()$ creates and returns a new object each time it is called,[4] and that $F_j()$ does not store the object before returning it, then $D'_{R_{F_j}} \equiv$ the set of object creation expression(s) contained in $D_{R_{F_j}}$, each appended with the identifier that distinguishes one call to $F_j()$ from another.

- otherwise, $D'_{R_{F_j}} \equiv D_{R_{F_j}}$.

This would (in the above example) cause the expressions added to $D_{leftFrontTire}$ to be distinct from those added to $D_{rightFrontTire}$.

8.2.4 A Further Improvement: Handling Multiple Instances of the Aggregate Object

Recall that in the above analysis, V_c (eg, *leftFrontTire*) in one object is not distinguished from V_c in another object of the same class. This is not a problem if it is known that there is only one instance of the aggregate class; that is, there is only object in the program containing V_c.

However, more often there may be more than one instance of the aggregate class. Consider the following example:

```
car1 = new Automobile();
car2 = new Automobile();
temp = new tire();
car1.leftFrontTire = temp;
car2.leftFrontTire = temp;
```

In this case, the same tire is assigned as the left front tire of two different cars, indicating that *leftFrontTire* of class Automobile cannot be designated as an (exclusive) component. However, the above algorithms do not detect this violation, since they do not distinguish between the *leftFrontTire* of *car1* from the *leftFrontTire* or *car2*; flow analysis for each of the final two assignments listed above will add D_{temp} to $D_{leftFrontTire}$.

To generalize the above algorithms, a variable V_c can be designated as a component only if both of the following conditions are true:

[4]Suppose that $D_{R_{F_j}}$ only contains object creation expressions local to $F_j()$. Then, each function invocation returns a unique object.

1. the values assigned to V_c are distinct from the values assigned to other component variables; that is, ComponentIfSingleAggregateInstanceP(V_c), defined above;

2. the values assigned to V_c in one instance of its aggregate class must also be different from values assigned to V_c in another instance of that class (i.e. the *leftFrontTire* of one Automobile must be distinct from the *leftFrontTire* of another Automobile).

The second condition is not decidable in general but can be determined to be true for some common cases, where all assignments to V_c are of the following forms:

1. $V_x = O_z$; that is, V_c is always assigned a newly created component object

2. $V_x = F_j()$, where $F_j()$ is known to return a different component object each time it is invoked

3. all assignments to V_c are generated by an object creation expression in the current invocation of the function containing the assignment. As noted in the prior section, reaching definitions analysis can determine this in some cases.

8.2.5 Limitations of This Approach

One limitation of the above approach is that it doesn't distinguish among multiple assignments of or to the same variable. That is, for an assignment of the form: $V_x = V_y$, this approach adds all of D_{V_y} to D_{V_x}, even though only a subset of the expressions in D_{V_y} might actually generate objects (assigned to V_y) that reach the assignment of V_y to V_x.

A less conservative approach would be to distinguish among multiple assignments to a variable. Assign a unique identifier A_n to each assignment, and associate with each assignment A_n the set D_{V_y,A_n}, where V_y is the left-hand-side variable of the assignment. Define the gen sets for each assignment to include only the reaching definitions for the right hand side expression. For example, for the assignment $V_x = V_y$, the gen set is D'_{V_y}, where $D'_{V_y} = (\bigcup_{A_n \in ReachingDefinitionsOfV_y} D_{V_y,A_n})$.

When the analysis is completed, D_{V_y} is computed as the union of the sets for all places where the variable may be assigned; that is, $D_{V_y} = (\bigcup_{A_n \in AssignmentsToV_y} D_{V_y,A_n})$.

Another limitation is that the above algorithm does not handle representations other than component member variables for denoting component/aggregate relationships. For example, in one proposed new design for the *Choices* Virtual Memory system, instead of storing a component MemoryObjectCache as a member variable of MemoryObject, a hash table has been proposed for storing all component/aggregate relationships in one place. The above algorithms would not handle this.

As with all data flow algorithms, the algorithms proposed here are conservative. Whether they are powerful enough to be useful can only be learned by experience. There probably are more powerful dataflow algorithms that could be invented for this problem, but that is not the focus of this thesis.

8.3 Adding A Member Variable To The Set Of Component Variables

This refactoring adds a member variable to the set of component variables for its class.

The argument to this refactoring is:

- the member variable (V) to be added to the component list.

After checking its precondition, this refactoring:

1. adds V to containingClass(V).SetOfComponents.

8.3.1 Preconditions

The precondition of this refactoring is:

1. qualifiesAsComponentP(member_variable).

8.3.2 Behavior Preservation

This refactoring is behavior preserving because it merely annotates design information to the program.

8.4 Removing A Member Variable From The Set Of Component Variables

This refactoring removes a member variable from the list of component variables for its class. the argument to this refactoring is:

- the member variable (V) to be removed from the component list.

This refactoring:

1. removes V from containingClass(V).SetOfComponents.

8.4.1 Preconditions

None.

8.4.2 Behavior Preservation

This refactoring is behavior preserving because it merely deletes design information from the program.

8.5 Moving Members into a Component (Pointer to Aggregate Stored in Component)

This section and section 8.6 define refactorings for moving a set of members from an aggregate class to the class of one of its components. One issue in moving such members is whether they can still reference members remaining in the aggregate class. The two refactorings differ in how they handle references *by* a member being moved *to* a member that will remain in the aggregate class. This section describes a refactoring where a pointer to the aggregate object is stored in the component, so that references back to members of the aggregate object can be made via this pointer. The next section describes a refactoring where the aggregate object is passed as an argument in calls to all functions that reference members remaining in the aggregate class.

During refactoring, the new members of the component class are added before the old members are deleted from the aggregate class. Sometimes, a component class will be a superclass (or subclass) of an aggregate class. For example, in a user interface framework the class Window might include components of class Sub-Window; these two classes would probably be part of the same inheritance hierarchy. In this cases, naming conflicts could occur when the members are added to the component class. When potential naming conflicts are detected, the members being added to the component class are given an (automatically generated) new name. After the refactoring is finished, the user could rename the moved members using the *change_variable_name* or *change_member_function_name* refactoring.

The arguments to this refactoring are:

- the aggregate class (C)
- the set of member variables being moved (setOfMovingVars)
- the set of member functions being moved (setOfMovingFuncs)
- the component member variable (V)

In this description, the class of the component member variable (ie V.type) is referred to as componentClass.

After checking its preconditions, this refactoring:

1. makes *AggregateObject* a member variable of componentClass, with a type that accepts an instance of class C:
    (a) if the member variable *AggregateObject* is already a member of componentClass, then:
        i. if the current type of *AggregateObject* is not C (or a superclass of C), change_type is called to change the type of *AggregateObject* to be a superclass common to C and the current type of *AggregateObject*.
    (b) else, if *AggregateObject* is already a member of one or more subclasses of C:
        i. for each subclass where the locally defined *AggregateObject* has a type that is not C (or a superclass of C), change_type is called to change the type of *AggregateObject* to be a superclass common to C and the current type of *AggregateObject*.
        ii. move_member_variable_to_superclass is called to move *AggregateObject* to componentClass.
    (c) else, create_member_variable is called to add the variable *AggregateObject* (type: the aggregate class) as a protected member of componentClass.
2. if access to V is not already abstracted, calls abstract_access_to_member_variable to abstract access in C to V.

3. in the update function (created in the previous step), adds the assignment:

   ```
   AggregateObject = <'this', the aggregate object>
   ```

   (steps two and three are needed so that all objects assigned to V will contain a pointer to the aggregate)

4. for each variable in setOfMovingVars, calls create_member_variable to add an equivalent (possibly renamed) variable as a public member of componentClass.

5. for each function in setOfMovingFuncs, calls create_member_function to add the signature of the (possibly renamed) function as a public member of componentClass. Then, for each function in setOfMovingFuncs, calls add_function_body to add the function body of each function being moved. References to members that will remain in the aggregate class are prefixed with *AggregateObject*.

6. replaces each reference *to* a member being moved, with a reference to its replacing member in componentClass.

7. calls delete_member_functions and delete_unreferenced_variable to remove the now unreferenced members from V.

8.5.1 Preconditions

The preconditions for this refactoring are:

1. qualifiesAsComponentP(V)
   (V qualifies as a component member variable of C.)[5]

2. $\forall$ F $\in$ setOfMovingFuncs,
   ```
       ∀ referencedMember ∈ (variablesReferencedBy(F) ∪
           functionsCalledBy(F)),
                   if referencedMember ∈ membersOf(componentClass) ∧
                       referencedMember ∉ (setOfVars ∪ setOfFuncs),
                           referencedMember.accessControlMode = public.
   ```
   (if any functions in setOfFuncs reference members remaining in componentClass, those referenced members are public).

3. the value of V is assigned before any member $\in$ (setOfMovingVars $\cup$ setOfMovingFuncs) is referenced; V is not reassigned. This is needed to ensure that all references to each moving member will point to the same location at all times. Program flow analysis would be needed to determine this.

4. $\forall$ class $\in$ (componentClass $\cup$ subclassesOf(componentClass)),
   ```
       memberVariableNamed(class, AggregateObject) ⇒
           commonSuperclass(memberVariableNamed(class, AggregateObject), C).
   ```
   (if *AggregateObject* is a member of componentClass or one its subclasses, the type of *AggregateObject* shares a common superclass with C (the aggregate class in this refactoring)).

[5]The requirement here is that the object assigned to V in one instance of C is not aliased as the value assigned to V in another instance of C. This condition is implied if V qualifies as a component variable. For further discussions, see section 8.8.

8.5.2 Behavior Preservation: Steps 1-5

For step 1-(a): for change_type: precondition 1 (assignments to the variable would remain type safe) is satisfied because the new type would be a supertype both of C and of its current type; precondition 2 (regarding assignments of the variable) is satisfied because by convention the variable is only used as a reference pointer, and is not reassigned. For step 1-(b): the preconditions of change_type are satisfied for the same reasons as given for step 1-(a). For move_member_variable_to_superclass: precondition 1 (regarding matching attributes) is satisfied by the prior step (when change_type is applied); precondition 2 (that the variable is not already defined in the superclass) by the conditions of step 1-(b). For step 1-(c): for create_member_variable: precondition 1 (the new variable doesn't collide with another variable referenced within its intended scope) is trivially satisfied because this step is invoked only when the variable is not defined in a superclass or subclass. Also, by convention, *AggregateObject* is a reserved name only used as a member variable to point to a component object's aggregate; therefore there will not be any local or global variables that collide with it.

For abstract_access_to_member_variable (step 2): precondition 1 (access to the variable has not already been abstracted) is satisfied because step 2-(a) is only invoked when this condition is true.

For the assignment added to the update function in step 3, program property seven (semantically equivalent references and operations) is preserved because the statement assigns a value to a newly created and unreferenced variable. By preconditions 7 and 8, and steps 1-(a) and 1-(b), the variable *AggregateObject* will be defined in the component class and its type will accept an instance of the aggregate class. Other program properties are trivially preserved.

For create_member_variable (step 4): precondition 1 (the new variable doesn't collide with another variable referenced within its intended scope) is satisfied because it is given a unique name in cases where a name conflict would occur.

For create_member_function (step 5): preconditions 1 through 4 (regarding collisions with members defined in superclasses or subclasses) is satisfied because it is given a unique name in cases where a name conflict would occur; precondition 5 is trivially satisfied because there is no function body.

For add_function_body (step 5): precondition 1 (all variables and functions referenced in the function body are visible from the superclass) is satisfied by precondition 2 of this refactoring; precondition 2 (the member function is not called) is satisfied because the function is newly added to the component and is not yet referenced.

8.5.3 Behavior Preservation: Step 6 (Replacing References)

The key step in the refactoring is step 6, where references to members in the aggregate class are replaced by references to members in the component. There are several things that might happen that could prevent this from being done safely:

Each new member might not be visible in all places where the old member (it replaces) was visible. However, in steps four and five the new members are public.

One or more of the functions being moved might make references to variables and functions that are visible from the aggregate class but not from the component class. By precondition 2, such references will be visible, provided that a pointer to the aggregate object is stored in the component; by step 3 of this refactoring, such a pointer is assigned.[6]

The type of the new member might be different from the type of the old member, and hence not support the same operations. This will not happen (by construction) since in steps 4 and 5 the new members have the same types as the members they are replacing.

Member variables in the component are referenced through a level of indirection (the component variable); this could lead to problems:

1. if an instance of the component class if not assigned to the component variable before a reference is made through that variable, the reference will be invalid. Precondition 3 ensures this will not happen.

2. if the value of the component variable changes (i.e. a new object is assigned to it) between references made via the component variable, the references will not be consistent (i.e. adjacent reads may not return the same value, and a read may not return the most recently written value). Precondition 3 ensures this will not happen.

3. if two different aggregate objects use the same indirection (i.e. point through the same component object), then updates made by one aggregate object will not be distinct from updates made by the other aggregate object. Precondition 1 ensures that this will not happen.

At this point, it has been argued that variables and functions will be visible to their callers before and after the move, that the types of members will be the same, and that, despite the level of indirection, references to the old member(s) in the aggregate class will consistently be replaced by references to the new members in the component. Below is a more formal argument. It considers the case of moving a single variable from an aggregate class to the class of one of its components. It can trivially be extended to handle multiple members.

Consider a program execution trace as a series of reads and writes to memory. For behavior to be preserved, the execution traces before and after the refactoring must be equivalent. For the traces to be equivalent, equivalent operations must be performed on equivalent values.

Regarding equivalent operations, there is a function that maps *each step* in the first trace (before the refactoring) to an *equivalent set of steps* in the second trace. Usually, a step in the first trace maps to a single step in the second trace, but not always. The refactoring will add in an extra level of indirection (the component) for some references, requiring an extra read operation.

Regarding the values read or written, they are either numbers or addresses of objects. Although equivalent values in equivalent traces will be identical if they are numbers, addresses may not be identical since the refactoring changes the sizes of objects. Instead, there is a function that maps variables and functions in one trace to their equivalent variables and functions in the other trace. One way to create this function is to say that the n'th new address referenced as a value (i.e. not just as a member variable) of one trace will be mapped to the n'th new address referenced as a value of the other.

Lemma: If traces are equivalent then equivalent locations contain equivalent values.

[6]For the refactoring in the next section, such references will be visible because a pointer to the aggregate object is passed in calls to functions that reference members not otherwise visible to the component.

Definition: The i'th member variable of equivalent objects are equivalent except for the variable being moved from an aggregate to a component. In this case, the variable in the aggregate in the first trace is equivalent to the variable in the component in the second.

Theorem: Traces before and after the refactoring of moving a variable from a container to a component are equivalent.

Proof: By induction. Base case is the empty trace. Induction step is to consider a trace of n steps. By the lemma, after the instruction corresponding to the n'th instruction of the first trace, equivalent locations contain equivalent values. Due to the level of indirection introduced by moving a variable to the component, a single instruction in the old trace may map to several instructions in the new trace, so the trace of n steps in the old trace corresponds to m ($\geq$ n) steps in the new trace. The n'th step might be a read, write, or a function call.

If the step does not reference the variable being moved, then in the new trace it maps to the same instruction applied to an equivalent location in the old trace. If it is a read operation, then by the induction assumption the location in the old trace will contain the same value as the equivalent location in the new trace, therefore the reads are equivalent. If it is a write operation, since the location in the old trace is equivalent to the location in the new trace, and the values written are the same, the writes are equivalent. If it is a function call, an equivalent function is called in each case, and the values passed in the call will be equivalent; therefore, the function calls are equivalent. Therefore, the traces are equivalent after the n+1'st step.

If the step does reference the variable being moved, in the new trace there is a level of indirection that will result in a set of steps in the new trace corresponding to the single step in the old trace. However, after the indirection the address read from, written to, or branched to in the new trace will be equivalent to the address in the old trace, and the operations and values are equivalent, therefore the traces are equivalent after the n+1'st step.

8.5.4 Behavior Preservation: Step 7

For delete_member_functions (step 7): precondition 1 (the function to be deleted is unreferenced or redundant) is satisfied because, as a result of the prior step, the function is redundant.

For delete_unreferenced_variable (step 7): precondition 1 (the variable being deleted is unreferenced) is satisfied because, as a result of the prior step, the variable is unreferenced.

8.5.5 Behavior Preservation: Summary

Each of the steps (1-7) of this refactoring are behavior preserving, therefore the refactoring is by construction behavior preserving.

8.6 Moving Members into a Component (Aggregate Passed as Function Argument)

This and the prior section define refactorings for moving a set of members from an aggregate class to the class of one of its components. This section describes a refactoring where the aggregate object is passed as an argument in calls to all functions that reference members remaining in the aggregate class.

The arguments to this refactoring are:

- the aggregate class (C)
- the set of member variables being moved (setOfMovingVars)
- the set of member functions being moved (setOfMovingFuncs)
- the component member variable (V)

For each member in (setOfMovingVars ∪ setOfMovingFuncs), a new name may be specified. This is needed to avoid naming conflicts that could arise when the aggregate and component classes share an inheritance relationship.

In this description, the class of the component member variable (ie V.type) is referred to as componentClass.

After checking the preconditions, this refactoring:

1. for each variable in setOfMovingVars, calls create_member_variable to add an equivalent (possibly renamed) variable as a public member of componentClass.
2. Let *FunctionsReferringBackToAggregate* be the subset of functions contained in setOfMovingFuncs that refer to variables and functions visible from C but not from componentClass.

   For each function in setOfMovingFuncs, this refactoring calls create_member_function to add the signature of an equivalent (possibly renamed) function as a public member of componentClass. (For functions in *FunctionsReferringBackToAggregate*, an additional argument is included, that points to the aggregate object.)

   Then, this refactoring calls add_function_body to add the function body of the newly created member functions. For each function in *FunctionsReferringBackToAggregate*, references only visible through the aggregate object are prefixed with the name of the new argument variable that points to the aggregate.
3. replaces each reference to a member in (setOfMovingVars ∪ setOfMovingFuncs) with a reference to the corresponding member in componentClass. Some function calls will pass a pointer to the aggregate object as an additional argument.
4. calls delete_member_functions and delete_unreferenced_variable to remove the unreferenced members from C.

8.6.1 Preconditions

The preconditions for these refactorings are the same as preconditions 1-3 of the prior refactoring.

8.6.2 Behavior Preservation

For create_member_variable (step 1): precondition 1 (the new variable doesn't collide with another variable referenced within its intended scope) is satisfied by precondition 3 of this refactoring.

For create_member_function (step 2): preconditions 1 through 4 (regarding collisions with members defined in superclasses or subclasses) is satisfied by precondition 3 of this refactoring; precondition 5 (all referenced variables and functions are reachable) is satisfied by precondition 4 of this refactoring.

For add_function_body (step 2): precondition 1 (all variables and functions referenced in the function body are visible from the superclass) is satisfied by precondition 4 of this refactoring; precondition 2 (the member function is unreferenced locally in the superclass or the superclass is abstract) is satisfied by precondition 3 of this refactoring.

The key step in the refactoring is step 3, where references to members in the aggregate class are replaced by references to members in the component. Behavior is preserved in section 8.5.3 for step 5 of the prior refactoring.

For delete_member_functions (step 4): precondition 1 (the function to be deleted is unreferenced or redundant) is satisfied because, as a result of the prior step, the function is redundant.

For delete_unreferenced_variable (step 4): precondition 1 (the variable being deleted is unreferenced) is satisfied because, as a result of the prior step, the variable is unreferenced.

Each of the steps of this refactoring are preserving, therefore the refactoring is by construction behavior preserving.

8.7 Moving Members into Aggregate(s)

This refactoring adds members to an aggregate class, to replace members in one of its components.

The arguments to this refactoring are:

- the aggregate class (C)
- the component member variable (V)
- the set of member variables in componentClass (V.type) being replaced in the aggregate class (setOfVars)
- the set of member functions in componentClass being replaced in the aggregate class (setOfFuncs)

After checking its preconditions, this refactoring:

1. for each variable in setOfVars, calls create_member_variable to add an equivalent (possibly renamed) variable as a public member of componentClass.
2. for each function in setOfFuncs, calls create_member_function to add the signature of the (possibly renamed) function as a public member of componentClass. Then, for each function in setOfMovingMembers, calls add_function_body to add the function body of each function being moved. References to members that remain in componentClass are prefixed with V.

3. for instances of C, replaces each reference *to* a member in (setOfVars $\cup$ setOfFuncs) with a reference to its replacing member in C.

4. for each member of (setOfVars $\cup$ setOfFuncs) that is now unreferenced,[7] calls delete_member_functions or delete_unreferenced_variable to remove it.

8.7.1 Preconditions

The preconditions for this refactoring are:

1. qualifiesAsComponentP(V)
   (V qualifies as a component member variable of C.)[8]

2. $\forall$ F $\in$ setOfFuncs,
   ```
       ∀ referencedMember ∈ (variablesReferencedBy(F) ∪
           functionsCalledBy(F)),
               if referencedMember ∈ membersOf(componentClass) ∧
                   referencedMember ∉ (setOfVars ∪ setOfFuncs),
                       referencedMember.accessControlMode = public.
   ```
   (if any of the functions being added to C reference members remaining in componentClass, those referenced members are public).

3. for all instances of componentClass assigned to an instance of C, all references to (setOfVars $\cup$ setOfFuncs) are made via V. Program flow analysis would be needed to determine this.

8.7.2 Behavior Preservation

For create_member_variable (step 1): precondition 1 (the new variable doesn't collide with another variable referenced within its intended scope) is satisfied because it is given a unique name in cases where a name conflict would occur.

For create_member_function (step 2): preconditions 1 through 4 (regarding collisions with members defined in superclasses or subclasses) is satisfied because it is given a unique name in cases where a name conflict would occur; precondition 5 is trivially satisfied because there is no function body.

For add_function_body (step 2): precondition 1 (all variables and functions referenced in the function body are visible from the superclass) is satisfied by precondition 2 of this refactoring;

[7]Instances of componentClass might be assigned to variables other than V. The members can not be deleted unless they are not referenced for all instances of componentClass.

[8]The requirement here is that the object assigned to V in one instance of C is not aliased as the value assigned to V in another instance of C. This condition is implied if V qualifies as a component variable. For further discussions, see section 8.8.

precondition 2 (the member function is not called) is satisfied because the function is newly added to the component and is not yet referenced.

The key step in the refactoring is step 3, where references to members in a component are replaced by references to members in the aggregate class. There are several things that might happen that could prevent this from being done safely:

Each new member might not be visible in all places where the old member (it replaces) was visible. However, by precondition 3 all references to the old members are made via V, which implies that all references must either be from within the aggregate object or via the aggregate object. By steps one and two, the new members are public members of the aggregate class and hence are visible in all places where the old members were referenced.

One or more of the functions being moved might make references to variables and functions that are visible from the component class but not from the aggregate class. However, by precondition 2 such references will be visible.

The type of the new member might be different from the type of the old member, and hence not support the same operations. This will not happen (by construction) since in steps 1 and 2 the new members have the same types as the members they are replacing.

If two different aggregate objects (both of class C) assigned the same component object to their member variable V, then with this refactoring some members that previously were shared between the two aggregate objects would no longer be shared. However, by precondition 1 each component object is exclusive to one aggregate object, so this will not happen.

At this point, it has been argued that variables and functions will be visible to their callers before and after the move, that the types of members will be the same, and that there is a one-for-one mapping of the old members to the new members that are replacing them.

As argued in section 8.5.3, the program traces before and after the refactoring are equivalent. References *by* the members added to the aggregate, *to* members of componentClass not being moved, will include an extra level of indirection (ie V). Conversely, references *to* the members added to the aggregate (other than those made *by* the members being moved) will have a level of indirection removed. These replacing references will be equivalent to the replaced references, and other steps will be the same, so the program traces are equivalent before and after the refactoring.

In step 4, the precondition for delete_member_functions and delete_unreferenced_variable is satisfied because these refactorings are only applied if the variable (or function) is unreferenced.

Each of the steps (1-4) of this refactoring are behavior preserving, therefore the refactoring is by construction behavior preserving.

8.8 Discussion: Components and Refactoring (Revisited)

The refactorings in sections 8.5 through 8.7 have as a precondition that the member variable into (or out of) which members are moved must be an *exclusive component*, as defined in section 8.2. As noted in a footnote for each of these refactorings, a less stringent precondition would be sufficient: that the instance assigned to the member variable in one aggregate object be distinct from the instance assigned to the same member variable of another aggregate object of the same type. This condition is implied if the member variable is an exclusive component, but is nonetheless a less stringent precondition.

This raises a question: is the precondition regarding exclusive components perhaps too restrictive for refactorings involving aggregates and components?

Not in general. More powerful refactorings could be written involving aggregates and components would require other conditions implied by exclusive components. Here are two examples:

1. For cases where an aggregate object contained several components that were instances of the same component class, a refactoring could be defined that replaced several members in the aggregate class with a single member added to the component class.

   For example, suppose the Automobile class contains the component member variables *leftFrontTire*, *rightFrontTire*, *leftRearTire* and *rightRearTire*, all of which were assigned instances of the Tire class. The Automobile class also contains four member variables that record, respectively, the warranty expiration for each of the tires. The refactoring would add a single member variable *warrantyExpiration* to the Tire class, replacing the four member variables in the Automobile class that record tire warranty expirations. This requires, however, not only that the instance assigned to each component member variable (say, *leftFrontTire*) in one car be distinct from instance assigned to the same member variable (*leftFrontTire*) in another car, but also that it be distinct from the objects assigned to other component member variables representing tires.

2. In cases where instances of the same class are components of several different classes, a refactoring could be defined that moves a variable out of the component class into the class of each of its aggregate classes.

   For example, while the Automobile class may hold an instance of the Engine class as a component, the Truck class may also hold another instance of the Engine class as its component. Moving *milesPerGallon* out of the Engine class means adding it to both the Automobile and Truck classes.

   For this refactoring to be behavior preserving, each instance of the component class must be assigned to a distinct aggregate object. This is implied if the component is exclusive.

In summary, the precondition regarding exclusive components, while more stringent than is necessary for the refactorings in sections 8.5 through 8.7, is nonetheless useful for these and other more powerful refactorings that could be developed regarding aggregates and components.

8.9 Converting an Association, Modeled Using Inheritance, into an Aggregation

8.9.1 Motivation

Consider the inheritance hierarchy shown in Figure 8.1. The class GraphicalObject defined functions for displaying itself and determining whether a particular point (location) was inside the object. Several subclasses were defined, the most recent being ArchitecturalObject. ArchitecturalObject needed the functions provided by GraphicalObject and was initially made a subclass of it.

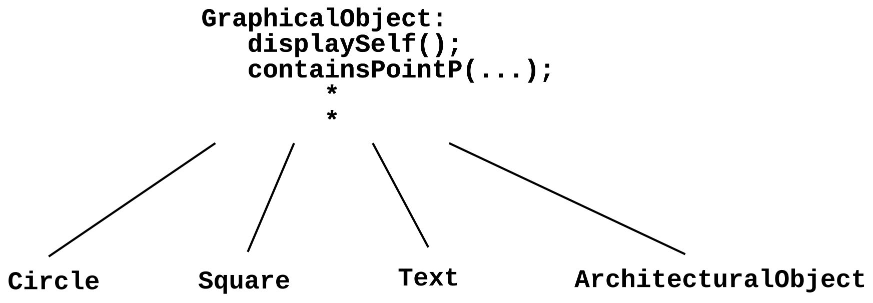


Figure 8.1: Class GraphicalObject and its Subclasses

The designer then realized that an ArchitecturalObject may change its shape, based on perspective or a new architectural insight. An ArchitecturalObject represents both an abstraction and its implementation; its shape is better represented as a component.

Thus, a more desirable representation would be as shown in Figure 8.2.

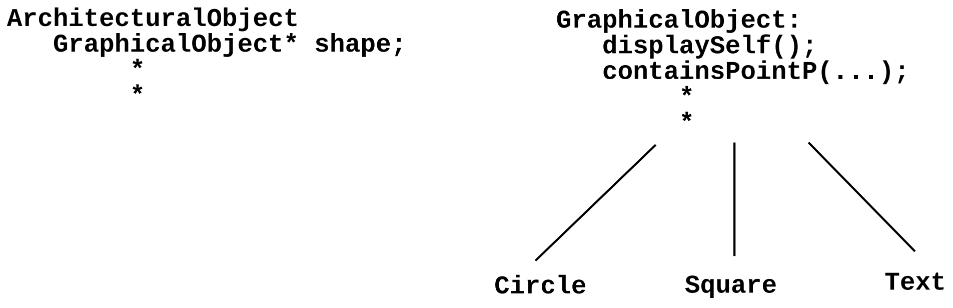


Figure 8.2: GraphicalObject and ArchitecturalObject, After Refactoring

This is an example of the refactoring defined in this section. Other examples are described in sections 3.4 and 9.2.

8.9.2 Arguments & Operations

This refactoring converts an association, modeled as a subclass/superclass relationship, into an aggregation. All behavior inherited by the subclass/client from its superclass before the refactoring will be delegated to a new component, which is an instance of the old superclass.

After this refactoring, the move_class refactoring could be applied to move the subclass/client.

The arguments to this refactoring are:

- the superclass (Super)
- the subclass/client (Client)

In the descriptions below: the set of variables inherited by subclass/client is referred to as *setOfVars*; the set of functions inherited by subclass/client is referred to as *setOfFuncs*.

After checking its precondition, this refactoring:

1. generates a unique name for the new component variable, and calls the refactoring create_member_variable to add the component variable to subclass_client
2. calls the refactoring defined in section 8.3 to qualify the newly created variable as a component
3. calls the abstract_access_to_member_variable refactoring for all var $\in$ setOfVars which have not already been abstracted.
4. for each inherited member function (ie setOfFuncs $\cup$ the functions added in the prior step), calls create_member_function to copy it into the subclass
5. $\forall$ var $\in$ setOfVars, replace in var's accessing and updating functions (in subclass/client) the current function body (a reference or assignment to var) with a call to the corresponding function in the new component.
6. for each remaining function copied into subclass/client in step 4, calls the refactoring replace_statement_list_with_function_call to replace the current function body with a call to the corresponding member function of the new component.

8.9.3 Preconditions

Precondition:

1. $\forall$ F $\in$ setOfFuncs,
   $\quad\forall$ refdItem $\in$ (variablesReferencedBy (F) $\cup$
   $\qquad$functionsCalledBy(F)),
   $\qquad\quad$refdItem $\notin$ (Client.locallyDefinedMemberFunctions $\cup$
   $\qquad\qquad$Client.locallyDefinedMemberVariables).

   (there are no references among the members inherited by subclass_client to members locally defined in subclass_client.)

2. $\forall$ F $\in$ setOfFuncs,
   $\quad\forall$ refdItem $\in$ (variablesReferencedBy F),
   $\qquad$Client $\subset$ scopeOf(refdItem).

   (all variables referenced by *setOfFuncs* are visible to subclass/client (ie their access control mode is not private).)

3. $\forall$ F $\in$ setOfFuncs,
   $\quad$F.accessControlMode = public.

   (all functions in setOfFuncs are public).

8.9.4 Behavior Preservation

For create_member_variable (step 1): precondition 1 (the new variable doesn't collide with another variable referenced within its intended scope) is satisfied because the new variable is given a unique name.

For the refactoring defined in section 8.3 (step 2): the precondition is satisfied because the variable qualifies as a component. The value (object) is statically assigned to it, and is unreferenced elsewhere.

For abstract_access_to_member_variable (step 3): precondition 1 is satisfied because this refactoring is only applied on variables that have not already been abstracted.

For create_member_function (step 4): precondition 1 is satisfied because the locally defined function is redundant; preconditions 2-4 (regarding equivalent signatures and function bodies) are satisfied because the superclass function is being copied into the class; precondition 5 (regarding references in the function body) is satisfied because the function references:

- variables whose scope includes both the superclass and subclass/client (by precondition two of this refactoring)
- functions which are either inherited by, or are identically defined in, the subclass/client (because the new function is identical to the previously inherited function it replaces)

For step 5: for all vars in setOfVars, the only references to the variables are by the accessing and updating functions. Changing the function bodies to be calls to the corresponding function in the component consistently replaces all refs to vars in setOfVars with with refs to the corresponding variables in the component. By construction, the type of each var in setOfVars is the same as the type of the var that replaces it. Thus, as for similar refactorings described earlier in this chapter, this step is behavior preserving.

For replace_statement_list_with_function_call (step 6): precondition 1 (the called function is visible from the calling function) is satisfied by precondition 3 of this refactoring; precondition 2 (semantic equivalence) is satisfied because it replaces the body of the function with a call to a function (in the component) that has a semantically equivalent function body. By construction, the two functions perform the same operations and, from the prior step, all variable references are semantically equivalent.

Thus, each step is behavior preserving and, hence, the refactoring is behavior preserving.

8.10 Discussion

8.10.1 Related Work in OODB Schema Evolution

Kim [66] describes composite objects in the ORION object-oriented database system, which are similar to aggregate objects. Orion supported *composite references,* which are similar to components. ORION supported several types of composite references, one important distinction being *shared* and *exclusive* references. An object could have at most one exclusive (and no shared) references, or any number of shared references. In refactoring, in order to avoid information loss the key quality of a component is that it be *exclusive* to one aggregate object. Thus, all components involved in these refactorings are exclusive.

8.10.2 User Interface

Section 6.2 describes a multi-window display to assist in defining the protocol for an abstract superclass. A similar interface (see Figure 8.3) could be used in migrating members between aggregate and component classes. The top window displays the member variables of the aggregate class Automobile. The middle two windows display the members for two of its component classes, Engine and Tire. In this example, the user requests to move the member variable *warrantyExpiration* from the component Tire class to its aggregate class Automobile. The user enters the request by, for example, clicking on the member to be moved, and either dragging it from the component window to the aggregate window, or by selecting the operation from a pop-up display.

The tool recognizes that class Automobile has four Tire components, so four variables need to be added to the aggregate class. In the lower window, it prompts for the names of the four new variables. Furthermore, since there is already a variable defined in class Automobile called *warrantyExpiration*, the tool warns that the new variables must be named differently from it.

Such an interface could be extended to provide display entire aggregation hierarchies (aggregate objects, their components and sub-components), graphs of inheritance and aggregation hierarchies, and in inverted hierarchy where a component class could be displayed, followed by all aggregate classes containing an instance of the component class. Precondition checking (eg: to avoid name conflicts) could be built into the interface, as shown in Figure 8.3.

Recall that in chapter six heuristics could be applied to suggest what members that could be migrated to an abstract superclass. Similarly, heuristics might be applied here to determine which members should be moved and when an association modeled using inheritance should be converted to an aggregation. However, just as in chapter six, it is unlikely that these heuristics will be error-proof, and (as described above) a sophisticated user interface can make these refactorings pretty easy, so heuristics are not the focus of this chapter.

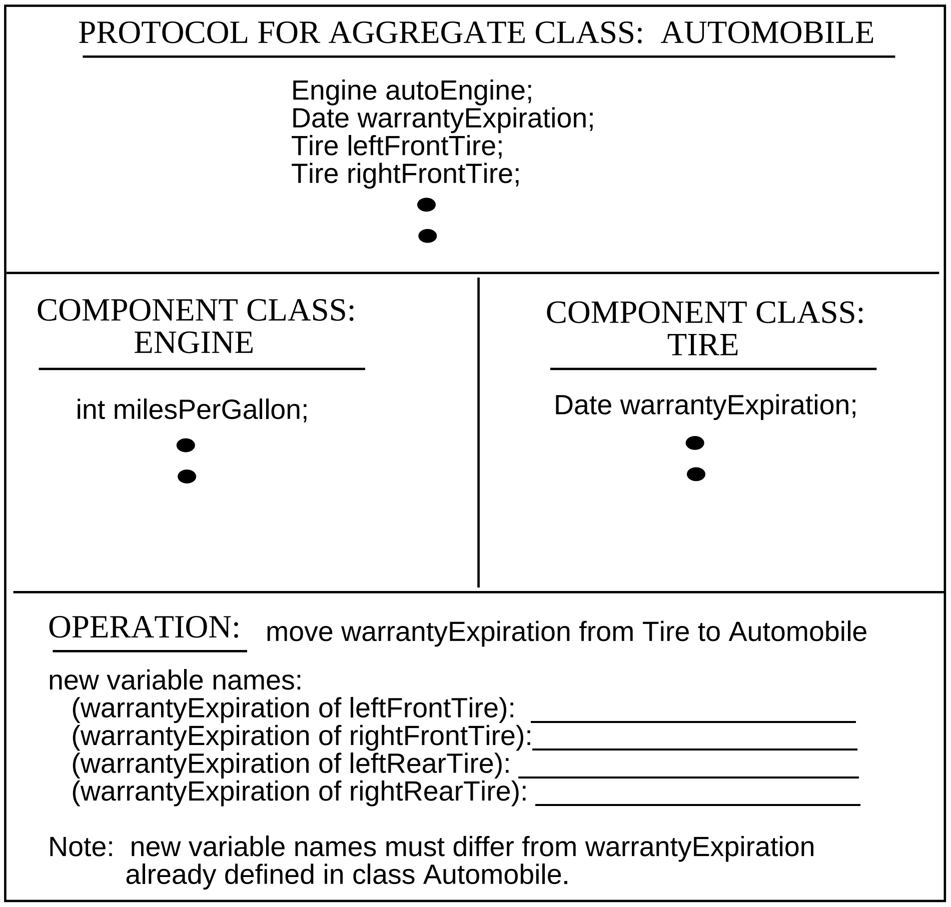

PROTOCOL FOR AGGREGATE CLASS: AUTOMOBILE
Engine autoEngine;
Date warrantyExpiration;
Tire leftFrontTire;
Tire rightFrontTire;
COMPONENT CLASS: ENGINE
int milesPerGallon;
COMPONENT CLASS: TIRE
Date warrantyExpiration;
OPERATION: move warrantyExpiration from Tire to Automobile
new variable names:
(warrantyExpiration of leftFrontTire):
(warrantyExpiration of rightFrontTire):
(warrantyExpiration of leftRearTire):
(warrantyExpiration of rightRearTire):
Note: new variable names must differ from warrantyExpiration already defined in class Automobile.

Figure 8.3: Migrating Members Between Aggregate and Component Classes

8.10.3 Summary

To summarize, this chapter defines several refactorings regarding aggregations:

- qualifying a member variable as a component
- moving members from an aggregate class to the class of one of its components
- moving members from a component class to the aggregate classes that contain component members which are instances of the component class
- converting a relationship, modeled using inheritance, into an aggregation.

This chapter defines how to check if a member variable qualifies as an exclusive component for its class, which is a precondition used in these refactorings.

Chapter 9

Examples

This chapter describes two examples of how refactorings can be integrated to improve the design of a program. The purpose of these examples is to clearly demonstrate how refactorings work, and to show that refactorings are not used in isolation, but rather that *sequences* of refactorings are often needed.

Chapter three described several *real* examples of refactorings manually applied to object-oriented application frameworks as they evolved. Those examples motivated the importance of refactoring in actual design tasks, but at the expense of some extraneous detail and complexity. By contrast, the examples presented here are simpler and clearer (albeit less practical).

The *Menu Planning* example demonstrates the use of refactorings defined in chapters six and seven for specializing a class and then migrating common code to a superclass. The *Matrix* example demonstrates the use of refactorings defined in chapters six and eight for changing how an interclass relationship is modeled (from inheritance to an aggregation); that example also includes refactorings for creating an abstract superclass of related component classes.

9.1 Generalizing & Specializing: Menu Planning Example

This example demonstrates the use of refactorings defined in chapters six and seven. This meal planning program assumes very predictable eating habits. The user specifies the days of the week that they will be in town, and the program prints out the dinner menu for that week. The implementation before refactoring (written in C++) is:

```
enum DaysInTheWeek { Monday, Tuesday, Wednesday, Thursday,
                     Friday, Saturday, Sunday};

class Date
   {
   protected:
      DaysInTheWeek dayOfWeek;
      int dayOfMonth;
   public:
      Date (DaysInTheWeek theDayOfWeek, int theDayOfMonth)
         {dayOfWeek = theDayOfWeek;
          dayOfMonth = theDayOfMonth;
```

```
              };
          void printDinnerMenu ()
             {if (dayOfWeek == Monday)
                cout << "Chicken Kiev";
              else if (dayOfWeek == Friday)
                cout << "Baked Flounder";
              else
                cout << "Stir Fried Vegetables";
             }
       };
     main()
       {
       Date firstDayInTown(Monday,1),
            secondDayInTown(Thursday,4),
            thirdDayInTown(Friday,5);
       firstDayInTown.printDinnerMenu();
       secondDayInTown.printDinnerMenu();
       thirdDayInTown.printDinnerMenu();
       }
```

In class Date, the member variable *dayOfWeek* is assigned an argument of the constructor, and is tested in a conditional statement of the member function *printDinnerMenu*. This conditional test suggests that the design of the program might be refined by specializing the Date class based on the enumerated values of the variable *dayOfWeek*.

The example below describes a special purpose, ultra-high level refactoring (ala macro) that performs its specialized task by combining the refactorings defined in prior chapters. When this ultra-high level refactoring is invoked, the refactoring tool (hereafter referred to as 'the tool') specializes the class Date based on the range of values (an enumerated type) of a member variable referenced by a conditional in the function *printDinnerMenu*. The refactoring is stylized, in that it uses the values of the enumerated type to automatically determine and name the subclasses.[1] It proceeds as follows:

Since the conditional tests the value of a variable whose type is the enumerated type *DaysInTheWeek*, the tool creates a subclass for each value in that enumerated type. Following the steps described in section 7.5, the tool creates seven subclasses of Date with the refactoring *create_empty_class*, one for each day of the week. The names of the subclasses correspond to the set of values of the enumerated type.[2] The tool assigns an invariant to each subclass of the form:

```
     dayOfWeek=<day of the week represented by the subclass>.
```

The tool defines a constructor in each subclass with the *create_member_function* refactoring. The constructor consists of an assignment that sets the value of the member variable *dayOfWeek*.

[1]Other such high-level refactorings could be defined, with minor variations.

[2]If the user wanted different names for the classes, they could be changed later using the *change_class_name* refactoring.

The tool then migrates the function *printDinnerMenu* down to the subclasses and simplifies it, as described in section 7.7. For example, the refactoring redefines the *printDinnerMenu* function in the subclass Monday as:

```
void printDinnerMenu ()
    {cout << "Chicken Kiev";
    }
```

The tool then specializes the instances of the superclass Date, using the refactoring defined in section 7.9.

Further refinements are possible. Since the class Date is truly abstract (it has no instances), and since the function *printDinnerMenu* is unreferenced elsewhere in the class and is redefined in all of the subclasses, the tool can delete the function from the superclass with the refactoring *delete_member_functions*.

Several further refactorings can remove unnecessary assignments, an argument variable, member variable and an enumerated type. These refactorings could be invoked individually by the user or, as is assumed here, invoked automatically.

First, the tool migrates to the subclasses the body of the constructor defined in class Date. Constructors in *C++* are different from other functions, in that a superclass constructor is not overridden by a subclass constructor; rather, when an instance of a subclass is created the superclass constructor is first called followed by a call to the subclass constructor. One way to think of this is that each subclass constructor begins with implicit calls to the constructors defined in its superclasses. Behavior can be preserved by adding the code from the superclass constructor to the front of the constructor in each direct subclass, while deleting the code from the superclass constructor.[3]

For example, the constructor in the class Monday would now be:

```
Monday (DaysInTheWeek theDayOfWeek, int theDayOfMonth):
  Date(theDayOfWeek, theDayOfMonth)
      {dayOfWeek = theDayOfWeek;
       dayOfMonth = theDayOfMonth;
       dayOfWeek = Monday;
      };
```

The first statement is a dead assignment, and the tool removes it. The argument *theDayOfWeek* is now unreferenced, and the tool removes it with the *delete_function_argument* refactoring. The member variable *dayOfWeek* is no longer referenced in *printDinnerMenu* since the conditional test has been removed; the final assignment statement in the constructor is thus a dead assignment and the tool removes it. Since *dayOfWeek* is unreferenced, the tool deletes it with the refactoring *delete_unreferenced_variable*. Then, the tool deletes the enumerated type *DaysInTheWeek* which is no longer referenced. The resulting code is now:

[3]Such a refactoring would be a slight variation on the *inline_function_call* refactoring.

```
class Date
   {
   protected:
      int dayOfMonth;
   public:
      Date (int theDayOfMonth) { };
   };

class Monday : public Date
   {
   public:
      Monday (int theDayOfMonth) : Date (theDayOfMonth)
         {dayOfMonth = theDayOfMonth;
           };
      void printDinnerMenu ()
         {cout << "Chicken Kiev";
         }
   };

(the remaining subclasses are similarly defined)

main()
  {Monday firstDayInTown(1);
   Thursday secondDayInTown(4);
   Friday thirdDayInTown(5);
   firstDayInTown.printDinnerMenu();
   secondDayInTown.printDinnerMenu();
   thirdDayInTown.printDinnerMenu();
   }
```

At this point, the Date class has been specialized and the conditional test removed. Refactoring could stop at this point.

However, by invoking two additional refactorings (described in chapter six), the implementation could be further optimized by migrating common code in *printDinnerMenu* function up to the superclass:

1. the function signature of the *printDinnerMenu* function could be added back into the superclass Date with the refactoring *add_function_signature*

2. common code in the subclass implementations of *printDinnerMenu* could be migrated up to the superclass, using the refactoring described in section 6.4.2. As part of that refactoring; the segments of differing code would be placed into a newly defined function, here called *mainCourse*.

The *printDinnerMenu* function is small, and automatically splitting up such a small function in this way would not improve the design in all cases. Thus, these latter two optimizations would probably be manually chosen, although the tool could suggest these changes to the user.

The refactored code is now:

```
class Date
   {
   protected:
      int dayOfMonth;
   public:
      Date (int theDayOfMonth) { };
      virtual char * mainCourse() = 0;
      void printDinnerMenu ()
         {cout << mainCourse();
         }
   };

class Monday : public Date
   {
   public:
      Monday (int theDayOfMonth) : Date(theDayOfMonth)
         {dayOfMonth = theDayOfMonth;
           };
      char * mainCourse ()
         {return "Chicken Kiev";
         }
   };

...
```

As a result of these refactorings:

- the design is clearer. The distinctions between days of the week are made more explicit. Existing instances have been specialized. In subsequent uses of the tool, instances would be specified according to the day of the week they represent.
- a conditional test, several assignments, variables and an enumerated type have been removed.
- the code for printing the menu has been encapsulated in one place, away from the code that determines the menu. To subsequently alter the means of displaying the menu, only the *printDinnerMenu* defined in class Date needs to be changed.

In summary, the result of these refactorings is a cleaner design, less code and possibly faster execution.

9.2 Converting to an Aggregation: Matrix Example

This example demonstrates the use of refactorings defined in chapters six and eight, focusing mostly on component related refactorings. Sometimes the relationship between two classes does not become clear until after implementation is underway. Consider the case where the class

TwoDimensionalArray is defined, then the Matrix class is defined as a subclass of it.[46] A matrix is a special use of a two dimensional array, and can be thought of as a specialization of it. For this example, matrices have a maximum of 400 cells each storing integer values.

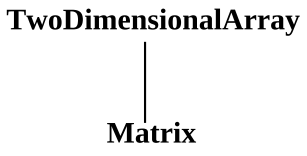


Figure 9.1: Inheritance Relationship Between Classes TwoDimensionalArray and Matrix

The classes are defined as follows:

```
class TwoDimensionalArray
   {
   protected:
      int elements[400];
      int columns, rows;
   public:
      TwoDimensionalArray (int numberOfRows, int numberOfColumns)
         { ... };
      virtual int get(int rowNumber, int columnNumber) { ... };
      virtual void put (int newValue, int rowNumber,
                        int columnNumber)
         { ... };
   };

class Matrix : public TwoDimensionalArray
   {
   public:
      Matrix (int numberOfRows, int numberOfColumns) :
        TwoDimensionalArray (numberOfRows,numberOfColumns) {};
      Matrix matrixMultiply (Matrix anotherMatrix)
        { ... j=get(x,y); ... put(k,x,y); ...};
      void rotate() { ...};
      Matrix martixInverse() { ...};
   };
```

It seemed reasonable at first to model this relationship using inheritance; the higher level matrix operations call the lower level functions inherited from its superclass. A matrix was thought of as a special type of two dimensional array. However, a matrix is a mathematical abstraction that can be represented in different ways. The representation is part of the matrix abstraction, but the matrix abstraction and its representation are really distinct concepts. Operations on the abstraction should be separated from operations on its representation. For applications where (for example) matrices are sparse, representations other than two dimensional arrays are more efficient.

The refactoring defined in section 8.7 is applied to better model the relationship between the classes Matrix and TwoDimensionalArray. During that refactoring, the tool:

1. adds an instance of the class TwoDimensionalArray as a member variable in class Matrix as member variable *matrixRepr*, using the create_member_variable refactoring.
2. qualifies the variable *matrixRepr* as an exclusive component of class Matrix, as described in section 8.3.
3. replaces all calls to the member functions *get* and *put* in class Matrix with calls to the corresponding functions in the component.

Then, the tool changes the superclass of class Matrix, using refactoring move_class.
The classes are now defined as follows:

```
class TwoDimensionalArray
   {
   protected:
      int elements[400];
      int columns, rows;
   public:
      TwoDimensionalArray (int numberOfRows, int numberOfColumns)
         { ... };
      virtual int get(int rowNumber, int columnNumber) { ... };
      virtual void put (int newValue, rowNumber, int columnNumber)
         { ... };
   };

class Matrix
   {
   protected:
      TwoDimensionalArray * matrixRepr;
   public:
      Matrix (int numberOfRows, int numberOfColumns)
        {matrixRepr = new TwoDimensionalArray(numberOfRows,
                                              numberOfColumns};
      Matrix matrixMultiply (Matrix anotherMatrix)
        { ... j=matrixRepr->get(x,y);
          ... matrixRepr->put(k,x,y); ...};
      void rotate() { ...};
      Matrix martixInverse() { ...};
   };
```

Note that the representation specific functions for retrieving and updating an element of the matrix are defined in the component class TwoDimensionalArray, whereas functions more closely related to the matrix abstraction are defined in the Matrix class. Note that in the *matrixMultiply* function of class Matrix, the *get* and *put* functions are referenced through the new component.

This separation not only clarifies the design by separating functions related to the abstraction and one if its possible representations, but also allows the program to be expanded to support multiple representations for multiple uses.

For some applications, the matrix is known to be sparse and representations more efficient than two dimensional arrays can be used. The following steps extend the tool to support matrix representations other than two dimensional arrays:

1. creating the class MatrixRepresentation as a superclass of TwoDimensionalArray, using the refactorings *create_empty_class* and *move_class*

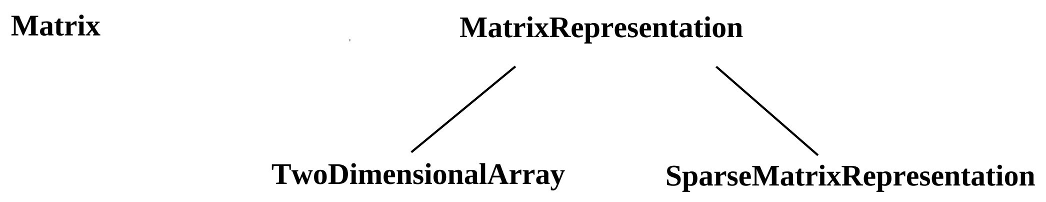
2. creating a new class SparseMatrixRepresentation as a subclass of MatrixRepresentation, using the *create_empty_class* refactoring

3. copying members from the class TwoDimensionalArray into the new class, and manually reimplementing those operations

4. migrating common behavior to the abstract superclass, using the refactorings described in chapter six

5. generalizing the type of the variable *matrixRepr* in class Matrix to be MatrixRepresentation.

The resulting inheritance hierarchies and class definitions are:



```
class Matrix
   {
   protected:
      MatrixRepresentation * matrixRepr;
   public:
      Matrix (int numberOfRows, int numberOfColumns)
        {matrixRepr = new TwoDimensionalArray(numberOfRows,
                                              numberOfColumns};
      Matrix matrixMultiply (Matrix anotherMatrix)
        { ... j=matrixRepr->get(x,y);
          ... matrixRepr->put(k,x,y); ...};
      void rotate() { ...};
      Matrix martixInverse() { ...};
   };

class MatrixRepresentation
   {
```

```
   protected:
      int columns, rows;
   public:
      MatrixRepresentation (int numberOfRows,
                            int numberOfColumns);
      virtual int get(int rowNumber, int columnNumber) = 0;
      virtual int put (int newValue, int rowNumber,
                       int columnNumber) = 0;
   };

class TwoDimensionalArray : public MatrixRepresentation
   {
   protected:
      int elements[400];
   public:
      TwoDimensionalArray (int numberOfRows, int numberOfColumns)
         { ... };
      int get(int rowNumber, int columnNumber) { ... };
      void put (int newValue, rowNumber, int columnNumber)
         { ... };
   };

class SparseMatrixRepresentation : MatrixRepresentation
   {
   protected:
      sparseArrayElement elements[50];
         /* maximum 50 elements for this sparse representation;
            type sparseArrayElement defined elsewhere */
      int get(int rowNumber, int columnNumber) { ... };
      void put (int newValue, rowNumber, int columnNumber)
         { ... };
   };
```

As a result of these refactorings:

- the mathematical abstraction is separated from the underlying representation
- the program defines multiple representations for Matrices, making explicit their common features. This structure would make it easier to extend the tool with additional matrix representations in the future.

9.3 Summary

This chapter describes two examples of how refactorings can be integrated. The first (*Menu Planning*) example demonstrates the use of refactorings defined in chapters six and seven for specializing a class and then migrating common code to a superclass. The second (*Matrix*) example demonstrates the use of refactorings defined in chapters six and eight for changing how an interclass relationship is modeled (from inheritance to an aggregation). These examples demonstrate how refactorings must be applied together to improve the design of a tool and support extensions.

Each refactoring checks is own preconditions and is guaranteed to preserve program behavior. Therefore, it is safe to string the refactorings together. This permits trying out different sequences of refactorings, and encourages design exploration.

It is expected that when a designer first uses a refactoring tool, they will manually apply a small sequence of refactorings to perform limited restructuring tasks. As the designer becomes more sophisticated, more complicated sequences of refactorings will be used; for a small set of frequently needed tasks, the designer will define higher-level refactorings (ala macros).

These examples demonstrate how the designer and the tool work together in improving the design of a program. The refactoring tool provides powerful support in performing precondition checking and implementing the changes. However, there is no way for a refactoring tool to be able to automatically predict what changes the designer will need and automatically do them. Therefore, the role of the designer remains important.

Chapter 10

Related Approaches

This chapter focuses on several areas of closely related work. In prior chapters, generally related work was described on the topics of software reuse, software restructuring, application frameworks, detecting code differences, and data flow analysis. This chapter discusses several approaches for supporting good object-oriented design style and for managing evolution of object-oriented systems. Also described is some recent work on restructuring applied to areas other than object-oriented systems.

10.1 CRC Approach to Designing Object-Oriented Systems

Wirfs-Brock, Wilkerson and Wiener [126] describe an approach to Object-Oriented Design focusing on *classes*, class *responsibilities* and the *collaborations* among classes.[1] In an initial exploratory phase, classes, responsibilities and collaborations are defined. In the subsequent analysis phase, systems are defined, collaborations simplified, hierarchies built containing abstract and concrete classes, and object protocols are defined.

While this approach is mainly targeted for the initial task analysis and system design, it has important implications on refactoring.

They emphasize the need for iterating the design of a system *before it is built.* This often involves some restructuring (on paper). By contrast, refactoring supports iteration and restructuring *after* (and *while*) the system is built.

They also note that a reusable design grows out of domain analysis, and the design must be clear in order to be reusable. Clarity is important not only before a system is built, but during and after it is built. While there are some general principles for improving clarity, some aspects of clarity are dependent on the application and the designer. Changes occur over time both to the designer's understanding of an application and in most cases to the application itself. Refactorings can assist in improving the clarity of a design, in light of these changes.

Since clarity is at least partly subjective, some interaction with the designer is needed during refactoring. The clarity of a design can be compromised if refactorings are applied "blindly" to optimize conformance to other design criteria, such as minimizing program size or minimizing access to variables.

[1]Other approaches to object-oriented analysis and design include [21] and [36].

10.2 Designing Reusable Classes

This refactoring research directly builds on research done at the University of Illinois by Johnson and Foote to support reusable classes [59]. To improving the design of an object oriented system, they define rules of thumbs for finding standard protocols, abstract classes and frameworks. Related to protocols, they recommend consistent naming of related functions, replacing case analysis with subclassing, and minimizing the size of methods and number of function arguments. Related to abstract classes, they recommend deep and narrow inheritance hierarchies, minimizing direct access to variables, and using subclassing for specialization. Related to frameworks, they recommend factoring implementation differences into subcomponents, s-plitting complicated classes and structuring the function call hierarchy so that subfunctions are defined in components. These rules of thumb are based mostly on experiences in Smalltalk programming.

Johnson and Foote focused at a high level on *what* ought to be done, while this research focused on *how* automated support could be provided. Their work described a set of high level guidelines that can be applied manually (or possibly automatically) to improve design, while this research precisely specifies behavior preserving abstract editing operations to automate design refinement. For example, one of their guidelines is: *"Split large classes."* This guideline is vague. This thesis defines several ways to split up large classes: by separating abstractions into abstract classes, by specializing classes using subclassing, and by defining components and moving members into the components.

This research differs from and extends [59] in other ways. Their paper includes a set of thirteen heuristics; this research defines a layered taxonomy of several dozen refactorings. The refactoring taxonomy made clear that their thirteen rules of thumb were not distinct but interrelate. Two of their rules are: *"Subclasses should be specializations."* and *"Split large classes."* (the latter, also mentioned above). Chapter seven describes an approach that exploits subclassing for specialization in splitting large classes. This research also uncovered design information that was needed to automate refactorings, and considered implementation issues.

Johnson and Foote recognized the need for this research in [59] and motivated its importance.

10.3 Achieving Good Style in Smalltalk Programs

Rochat [98] proposes several guidelines for achieving good programming style in Smalltalk. Among the guidelines: Carefully name classes and class members. Each class and method should have a single purpose; multi-purpose classes and methods should be split. Subclasses should be refinements (in protocol or implementation) of their superclasses. Nested conditionals should be replaced with multiple classes to achieve polymorphism.

Rochat concludes [98] that organizational tools are needed to support these guidelines. These guidelines are reflected both in the supporting refactorings and most clearly in the discussion of subclassing in chapter seven of this thesis.

10.4 Demeter Project

Lieberherr and others in the Demeter project at Northeastern University have studied how to improve the productivity of object-oriented designers and programmers[71, 69]. They have proposed rules for bringing discipline to the object oriented design process (the "Law of Demeter") and have developed a set of structures and tools, which they package as the Demeter system. Their style is claimed to increase information hiding, minimize coupling between classes, and improve maintainability and comprehensibility. They have developed tools to check if programs conform to the law.

There are some shortcomings to the Demeter approach. The Demeter researchers [72] note that restructuring a program to make it conform with their "law" can require increasing the number of methods and method arguments, and result in slower execution speed and poorer readability of code. Wirfs-Brock and Wilkerson [125] argue that the law overconstrains the connections between objects, and more importantly shifts the focus away from the responsibility-driven aspects of design (where it belongs) toward the data-driven aspects. Sakkinen [108] discusses other problems in applying the Law of Demeter.

Nonetheless, the Demeter work is an important contribution to object-oriented design style. Abstraction is important in improving the maintainability of object-oriented systems. A refactoring that directly supports the hiding of variables, in line with the Demeter guidelines, is *abstract_access_to_member_variable*. Another way to better encapsulate variable access is to refactor variables into a component and limit access to the component.

Another important contribution of their work is in defining structures and tools for object-oriented design. A *class dictionary* defines classes, members and inheritance relationships. Class dictionaries can be represented graphically, where classes and members are represented as vertices, inheritance relationships are represented using *alternation edges*, and class/member relationships are represented using *construction edges*. There are tools for converting between textual and graphical representations of class dictionaries, for uncovering subgraphs of related objects called *propagation patterns*, and for generated C++ source code from their representations.

Bergstein [15] defined a set of object preserving class transformations that can be applied to class dictionary graphs. He shows that all object preserving class transformations that can be performed on class dictionary graphs can be decomposed into a small set of primitive operations: adding or deleting a "useless" alternation, abstracting or distributing common parts and part replacement. These changes correspond to adding or deleting an unreferenced class as a subclass of an existing class, migrating members up or down an inheritance hierarchy, or changing the type (class) of a member. Each of these changes, provide that their preconditions are met, will preserve existing objects; that is, the set of members for each object will be unchanged by these transformations.

These operations correspond to some of the supporting refactorings defined in chapter five, which are frequently used to implement other refactorings. However, the refactoring approach overall is more general and powerful. Refactorings operations that cannot be expressed in terms of these transformations include abstracting variables, modifying functions and moving members between component and aggregate classes. Abstracting variables is not needed in the Demeter system, since variable references are abstracted in the code generated from their representations. Function modifications are not supported because the data dictionary graph is limited to supporting data modeling. Perhaps the most important difference between refactorings and the object preserving class transformations is that refactoring can change the set of members in a class, as long as the overall behavior of the program is preserved. This permits changes such as moving members between component and aggregate classes, which are not covered in Bergstein's approach. Extensions to Bergstein's approach are being investigated by the Demeter researchers [70].

10.5 Reorganizing Generic Applications: Ithaca

As part of the ITHACA (Esprit II) project, research by Casais at the University of Geneva investigated approaches to support the evolution of a generic application, which is similar to an application framework. As with Bergstein's approach described above, Casais' focus was upon inheritance related changes.

He defines four means for supporting evolution [31]:

- *tailoring*, where the existing class definitions are not directly modified, but adaptations are applied to inherited properties when deriving new subclasses. For example, when introducing a new subclass a function inherited from its superclass may be overridden. Or, some object-oriented languages allow a variable inherited from the superclass to be renamed in a subclass be retain other properties.
- *surgery*, which are structural changes made to a particular class. These changes include adding, removing or renaming a variable or function, or changing the type of a variable.
- *versioning*, to support configuration management of multi-person projects
- *reorganization*, which are more significant structural changes which may involve the introduction or suppression of classes.

Adding a new, unreferenced class is an example of tailoring. Many refactorings, such as adding a variable to a class, are examples of class surgery. The refactorings that move classes, and the three higher level refactorings, are examples of class reorganization.

Casais' research focuses on the final area (reorganization). His approach focuses on reuse by inheritance. A class specification is defined as a set of properties. These properties are either inherited or locally introduced. If classes or class "properties" (members) are arbitrarily added to a class hierarchy, classes might inherit properties that violate their specification, or properties may be reintroduced that could have been inherited from an existing class. Local and global reorganization strategies are introduced. When a new class is introduced, localized reorganizations to the class hierarchy can be applied so that the new class does not inherit properties that are rejected in by specification. At a more global level, the same property may appear in two otherwise unrelated classes. To avoid this duplication, the inheritance hierarchies are globally reorganized in a way that the shared property is moved to a third class and the two classes inherit from it.

Eliminating duplicated properties, and ensuring that the inherited behavior of a class doesn't violate its protocol, are useful design goals. However, these goals do not always contribute to, and can sometimes work counter to, the more important (albeit less rigorous) goal of refactoring, which is to *improve design clarity*.

Casais' strategies sometimes, *but not always*, will produce what the programmer intended. A central concept of application frameworks is that each class should correspond to a meaningful abstraction. However, the reorganizations noted above can introduce classes that do not necessarily correspond to meaningful abstractions, but rather are needed to make the mechanism work. Also, there is heavy use made of multiple inheritance, which can make it difficult to understand how inheritance hierarchies map into type specialization hierarchies.

Casais' research makes important contributions including the characterization of changes into the four categories listed above, his local and global reorganization strategies, and his object model. However, as he notes in [31], his reorganization algorithms deal exclusively with inheritance. There is no facility for splitting methods, restructuring their code, or changing the client/server relationship between classes. These areas are addressed in the refactoring research, as Casais cites in [31]. Splitting and restructuring functions is covered in chapter six. In chapter eight, an approach is described for replacing inherited behavior with behavior delegated to (provided by) a component.

10.6 Managing Schema Evolution in an OODB

Work in object-oriented databases (OODBs) has been motivated by several well-known shortcomings of more conventional database technology, one being that the structural information stored in a database schema cannot naturally model frequently useful semantic concepts, such as generalization and aggregation relationships. Schemas evolve over time; many of the changes made to schemas in OODBs are similar to refactorings.

Schema evolution has been studied for several OODB systems, most notably ORION [9, 66].[2] Just as with refactoring, the methodology for handling schema changes has included:

- developing a taxonomy of changes
- defining a set of properties that must remain invariant across schema changes
- defining the changes in the taxonomy in a way that preserves these properties.

.

In the ORION system, a distinction is made between *soft* changes (those that do not require existing instances to change their classes) and *hard* changes. Examples of soft changes are adding or renaming an entity, similar to some supporting refactorings. An interesting hard change is the partitioning of a class into several subclasses. One type of partitioning is horizontal partitioning, a database selection operation which is analogous to specializing a class into a set of subclasses. An existing instance is assigned to a subclass if it satisfies a "partitioning condition" which is a kind of class invariant. Another type of partitioning is a vertical partitioning, a database projection operation which is analogous to simplifying a complex aggregate class by defining components and migrating members to those components. One aspect of vertical partitioning that makes it different from migrating members to components is that, after the attributes have been moved to the new classes, and the original class is deleted.

ORION represents aggregations as *complex objects*. A *composite link* represents a relationship between a composite object and its component. A *composite link* can carry special semantics, such as the component is exclusive to one composite object. This notion of exclusive components was applied in the refactorings described in chapter seven.

Not surprisingly, the focus of research into OODBs has been on altering and migrating *data* (member variables) with less emphasis placed upon member functions. However, there are facilities in ORION to detect when functions (methods) have been invalidated because the data they reference has been altered due to a schema change.

Some research into OODBs has considered versioning issues and multiple inheritance, outside the scope of this refactoring research. Conversely, OODB research has not focused on

[2] Other systems include Gemstone [92], O_2 and OTGen. Casais [31] compares features of these object-oriented database systems with regard to schema evolution.

several topics important in refactoring, such as simplifying conditional expressions using invariants, and converting an association modeled using inheritance into an aggregation.

10.7 Program Transformations

Refactorings support change by performing a type of program transformation. Research into more ambitious forms of program transformations and automatic programming has been underway for over 15 years. Early work in this area was done by Burstall and Darlington [27]; extensive work has been done by Balzer and others at USC/ISI [8], including recent work into building an evolution transformation library[61].

This work has focused on the process of converting a formal specification into an efficient implementation. Examples of behavior preserving transformations are:

- *folding and unfolding.* Folding replaces a code segment with a function call; unfolding inline expands a function call. There are refactorings for each of these operations.

- *abstraction.* Abstraction substitutes a variable for every instance of an expression, and defines that variable to be the value of that expression. This transformation did not arise in the refactoring research, although there are analogous refactorings to create a function whose body is an expression, and convert an expression to a function call.

- *splitting type.* This is a transformation that splits a type into subtypes, analogous to the refactoring that specializes a class by defining subclasses.

- *bubble up.* This transformation moves a function out of an enclosing module and expands its scope. Refactoring can expand the scope of a function by changing its access control mode or by moving it to a superclass or to a component class.

Since program transformation systems begin with a formal specification rather than source code, they can by construction avoid some of the code level complexities and dependencies that can make refactoring source code difficult. On the other hand, it is difficult to correctly define the input to these systems. As Balzer notes in [8], one of the chief failures in program transformation systems has been the unreadability of their specification languages. It has taken a high level of user sophistication, both in the application domain and in the domain of specification languages and transformations, to apply many automatic programming approaches.

Refactorings are a type of program transformation, but with an important difference from many past transformational approaches. Those approaches usually assumed a top down process, going from a formal, abstract specification to an implementation. In contrast, refactorings are lower level, systematic, and are less dependent on the semantics of a program. The idea is to raise program editing to a higher level, not to build a design language.

Refactorings do not force upon a designer a rigid, top down design style. Experience has shown that the development of a large software program does not proceed in a purely top down manner [38, 89]. Software design is in many ways an informal process [63]. Refactorings support iterative conceptual shifts. For example, the refactoring that defines an abstract superclass generates a common design abstraction from low-level implementation; this design abstraction can then be used for further low level implementation. Refactorings shield the designer from many of the lowest level implementation details regarding program restructuring, and can make it easier to shift between high level design and implementation. Furthermore, refactorings support design without requiring the designer to be proficient in writing abstract, stylized formal specifications.

One important insight gained from research into program transformations is that while many systems started out with the goal of fully automating the transformation process, it became obvious that in handling non-trivial programs a transformation system required some interaction with the user. A similar insight has been gained from the refactoring research.

10.7.1 Transforming Block Structured Languages

Griswold [52] investigated meaning preserving transformations to restructure programs written in a block-structured programming language, to make maintaining such programs easier. The language analyzed in this research was Scheme. From the program source, a program dependency graph is created. A set of meaning preserving graph transformation rules, similar to local compiler optimizations, are defined for manipulating statements within a block. These rules include, for example:

- renaming a local variable
- adding a new variable to store an intermediate result
- replacing an expression with an equivalent expression (such as a variable, if the variable has previously been assigned the value of the expression, or a function call, if the function body is equivalent to the expression it replaces)
- converting a scaler into some nested structure (such as an array or record),
- moving an assignment into and out of a conditional block.

Like the refactoring research, his research investigated source to source program restructuring to aid software maintenance. However, his work [52] focused on transforming the syntactic constructs of a block-structured language; his focus was not on transformations involving inheritance or a type hierarchy. In [52] he cites our refactoring research and discusses how his approach might be applied for object-oriented systems. He notes several complexities that inheritance causes in program analysis and transformation; these complexities are dealt with more fully in the prior chapters of this (refactoring) thesis.

His use of program dependency graphs and flow laws for describing source to source transformations is an important contribution. One reason why he choose Scheme as a language to analyze was because a tool was available for generating and manipulating program dependency graphs. A area for future research would be to develop and investigate the use of such a tool for refactoring object oriented programs.

10.8 Summary

This chapter focuses on several areas of research closely related to refactoring. These areas have included supporting good object-oriented design style, managing the evolution of object-oriented systems, and restructuring applied to areas other than object-oriented systems.

Chapter 11

Conclusions

11.1 Summary of Contributions

This thesis defines a set of program restructuring operations (refactorings) specific to supporting the design, evolution and reuse of object-oriented application frameworks. It makes several important contributions:

1. It identifies a set of program restructurings (refactorings) that people apply to object-oriented application frameworks.
2. It shows how to automatically support refactorings in a way that preserves the behavior of a program. Most of the refactorings are simple to implement and it is almost trivial to show that they are behavior preserving. A small set of refactorings are much more complicated. A layered hierarchy of refactorings was defined to help deal with the complexity of the refactoring process.
3. It defines in detail three of the most complex refactorings: generalizing the inheritance hierarchy, specializing the inheritance hierarchy and using aggregations to model the relationships among classes. It decomposes these operations into more primitive parts, and discusses the power of these operations from the perspectives of automatability and usefulness in supporting design.
4. It defines design constraints needed in refactoring, specifically class invariants and exclusive components. These constraints are needed to ensure behavior is preserved across some refactorings. The thesis discusses how to analyze a program to determine if it satisfies these constraints, and how to use this design information to refactor a program.

11.2 Limitations of Approach

There are several limitations of this approach:

1. It requires behavior preservation at every step. There are cases where states of temporary inconsistency might be desirable, for example when replacing one algorithm with another.[1]

[1]Refactorings can, however, be applied to localize behavior within a class or small set of classes and thereby better encapsulate such changes.

2. The description of refactoring in this thesis is limited by the features in the underlying programming language being refactored. The strict inheritance relationship among types, a feature of *C++,* limits type substitution, generalizing and specializing. Multiple inheritance is not incorporated into the refactoring approach defined here, nor are language features such as automatic type conversion (casts in *C++*).

3. The power of this approach is also limited by the program analysis techniques used. The undecidability of the basic problem *requires* that any 'solution' be conservative. The algorithms presented in this thesis can almost certainly be improved upon. However, an improved solution will still be conservative; there will still be cases in which valid refactorings will be disallowed.

4. The practical utility of this approach is not known. A valuable contribution could be made by software engineering studies of refactoring large programs written in several practical (albeit complicated) languages with several different programming styles. Automated refactoring should probably be most valuable when applied to large programs, but global program analysis may be expensive when applied to these programs.

11.3 Implementation Considerations

Experimental research is needed, studying the role of refactoring in large, multi-person software developments. Software process studies are needed, to better understand the role of refactoring in concept prototyping, design refinement, implementation and maintenance tasks.

This section discusses some of the issues that would arise in implementing a practical refactoring system. This discussion is based on lessons learned from the prototype built as part of this research.

A practical refactoring system would have several requirements. Since refactorings are *behavior preserving* program restructuring operations, a practical refactoring system must provide a program representation that is rich enough to support precondition checking and perform consistent updates. This chapter discusses techniques for parsing a program, and for generating and maintaining cross reference information. To support design exploration, a system must also be flexible and fast. Finally, for a refactoring tool to be useful, it must integrate well with the organization targeted to use it.

11.3.1 Explicitly Representing the Structure of a Program

A textual representation of program source is not rich enough to support refactoring. Consider, for example, the refactoring that changes the name of a member function, defined in chapter five. This refactoring changes not only the function definition but also changes all places where the function is called. Depending on the access control mode of the member function, it may be referenced in its containing class, its subclasses and other places that reference an instance of one of those classes. This refactoring must change the function name in all these places. However, there are cases where a name substitution should not be made. The same name may be used elsewhere in the program to refer to a variable or possibly even to a class; distinguishing these cases requires an analysis of the program syntax. Or, a different function with the same name may be defined in another, unrelated class; distinguishing this case requires type information. A simple textual scan and replacement approach cannot make these distinctions.

Rather, a program representation is needed for refactoring that makes structural information explicit, and that supports automated analysis, better than program source. This structural information is needed not only to execute a refactoring, but also to check the preconditions that determine if the refactoring is valid. For example, precondition checking for the refactoring that changes the name of a member function involves collecting the sets of member functions defined on a class and its superclasses. This requires syntax analysis and type information.

An abstract syntax tree, annotated with type and reference information, is a representation can be used to refactor programs.

11.3.2 Software Refactoring Process

Current programming practice usually involves maintaining and updating programs in source code form. In order to integrate refactoring into such environments, program source must first be converted to an abstract syntax tree representation before refactoring, and the updated program source must be regenerated from the representation after refactoring. Figure 11.1 (at end of chapter) shows the stages in the refactoring process. Within a refactoring system, a parser creates a syntax tree representation of the program source. The parse trees are passed to the refactorer, which first analyzes the trees and adds type and cross-reference information. Then the refactorer manipulates the representation based on commands from the user (a human doing design or possibly a higher level program). Finally, a pretty printer is called to print the refactored code.

The approach that we used to build the prototype was to write a simple parser from C++ into Lisp forms using a YACC-compatible specification of C++ developed by Roskind [100]. A Common Lisp [62] program, making extensive use of the Common Lisp Object System (CLOS) facilities, used the output of the parser to build a complex structure representing the program. Each program part was represented as an aggregation, where the aggregate object corresponded to the root node of the parse tree and the child nodes were represented as components. Many components were themselves the root of a sub-tree, whose children were represented (recursively) as components. Each node in the tree was an instance of one of the (CLOS) classes defined in the refactorer corresponding to the significant non-terminals defined in the grammar. The refactorings were implemented as Common Lisp functions. A crude code regenerator, used mostly for debugging, was implemented as a CLOS generic output function, distributed among the classes.

Using an object-oriented approach to represent the parse tree as an aggregate object made the factoring of some functions easy. Many of the operations on for example a class definition involve traversing the entire tree representing the class, but only a few nodes were modified in the process. These operations were defined as CLOS generic functions, defined on all classes representing nodes in the tree. For most classes, these functions just consisted of calls to the corresponding functions in the components; the interesting modifications were encapsulated within the classes of the nodes being changed.

11.3.3 Generating and Maintaining Cross References

Refactoring one part of a program often requires changing other parts of the program that refer to it. It is sometimes necessary (especially during precondition checking) to find all references to a variable, function or class being changed. A refactoring system must detect these "backward references" (that is, references *to* the variable, function or class), and consistently change the program. If, for example, the refactoring that moves a variable from an aggregate class to the class of one of its components. The refactoring involves adding a new variable to the component class, deleting it from the aggregate class and changing references to the variable. When refactoring is finished, it is essential that no references to the old, deleted variable be left dangling in the system.

References made *by* a variable, function or class are most naturally resolved when that object is loaded; however, there are several approaches for handling references *to* these parts of a program.

At one extreme, this cross reference information can be precomputed and stored in a database, which is updated as the program structure changes. At the other extreme, cross reference information can be generated as needed by traversing the program structure.

Precomputing the cross reference information has the advantage of making cross reference lookup (queries) easy, but making updates to this cross reference information can be hard. Several tools have been developed to create these tables from C++ source code [50, 81]; tools have also been developed for other object oriented languages. However, refactorings change program structure, and almost every refactoring requires the cross reference database to be updated. These database changes can be costly and complex.

At the other extreme, generating the cross reference information as needed from the program structure has the advantage that the resulting cross reference information is always up to date. When a program is refactored, there is no need to also update a separate structure that records cross-reference information. However, tree traversals can be costly for large programs, and since cross reference information is needed at several points in most refactorings, for a large program this approach is probably unacceptably slow.

Approaches corresponding to these two extremes were tried in this research. The first approach made extensive use of a cross reference database. When the program was read in to the refactorer, several tables of semantic information were generated and stored within a special object called the *global information manager*. During refactoring, when the structure of the program was constantly changing, it was difficult to maintain consistency between the tables and the implementation.

Then, a second approach was tried, where this information was generated dynamically. The only reference information stored within the global information manager was the list of classes and global declarations (functions and enumerated types). Later, when a refactoring needed to know for example all references to a member variable, the global information manager generated the set by traversing the relevant parse trees. Compared to the earlier approach, this approach to handling cross reference information was simpler to implement and debug; working with small examples there were no noticeable performance penalties.

A practical refactoring system would need a method for handling cross reference information that falls somewhere between these two extremes. The cost of repeated tree traversals could be prohibitive if the program being refactored is large (eg: a million lines of code). One way to reduce the number of tree traversals needed would be to "cache" within a referenced object the set of objects (representing parts of the program) that might refer to it, as this information is often needed during refactoring. The cache of backward references to, for example, a variable could be implemented as (pointers to) the set of functions that *might* reference the variable.[2]

[2] The cache could, alternatively, be engineered to contain pointers to the classes or expressions that might reference the variable.

The cache would contain a *superset* of all true backpointers; in other words, it would point to all the functions that reference the variable, plus possibly some functions that no longer reference the variable. Since the set of backpointers is a superset of the set of true backpointers, the cache only needs to be updated when references to the variable are *added* to the program (not when they are deleted). The cache could be cleaned up when an attempt is made to find all references to the variable; pointers to functions that no longer reference the variable would be removed from the cache.

Clearly, for large programs the representation(s) used will influence how expensive it is to refactor the program. One way to achieve some efficiencies in maintaining consistency between multiple program representations is described by Griswold [52]. His restructuring of Scheme programs used an abstract syntax tree (AST) representation, and a program dependency graph (PDG) which recorded data flow and control flow dependencies. PDG updates were expensive. The AST announced three events as the program was restructured: insert, delete and change. The 'AST-PDG consistency' component received these events, and determined the updates needed to the PDG. A 'lazy consistency' strategy was applied, where updates to the PDG were cached and not actually applied to the PDG until after a transformation was completed and a new transformation was about to begin. Such techniques may apply for a practical refactoring system.

11.3.4 Supporting Design Exploration

As noted at the start of this thesis, design is hard. Designing reusable software involves creativity and exploring design alternatives. Speed and clarity of the user interface are important in a refactoring tool that supports design exploration. A refactoring tool that is too slow will "get in the way" and discourage trying out alternative designs. Ideally, a refactoring should execute instantaneously. Also, since a major purpose of refactoring is to make the design on the system clearer, and since clarity is in part subjective, it is important that a refactoring tool make clear to the user the effects of an operation. Since alternatives are sometime explored and then discarded, it is also useful to provide a mechanism that is flexible and can undo the effects of a refactoring.

Refactoring a large program could be very slow if it required repeatedly generating cross reference information that involved exhaustive tree traversals. As noted above, there are ways that a system could be engineered to reduce the cost of these queries. Fortunately, due to technology advances, issues of processing speed and memory size are becoming less of a concern than in the past. Performance issues will be better understood as practical refactoring systems are implemented and used.

Clarity in the user interface is important because, as discussed earlier, many refactoring tasks (especially the more complex refactorings) require some user interaction for the results to be useful. Rak [93] describes an interface for refactoring Smalltalk methods and migrating code to a common superclass; his approach is described and extended in chapter six. User interface issues regarding refactoring process is an area for future research.

Several approaches can be applied to "undo" the effects of a refactoring. Checkpointing, logging and traditional change management techniques can be applied to roll back the effects of changes. In some cases a refactoring (or short sequence of refactorings) can be applied to undo the effects of an immediately prior refactoring.

In summary, a practical refactoring system that supports design exploration needs to be fast, clear and flexible.

11.3.5 Other considerations

The value of a refactoring tool to an organization will be measured by how well it can help that organization develop software in the relevant application domains. While the refactorings described in this thesis are domain independent, the choice of refactorings to apply to a program will be motivated by the user's understanding of the domain and the state of the program. There are several implications that an application domain can have on refactoring. If the application has a well defined vocabulary of synonyms, this can make similarity detection easier when creating a common superclass. On the other hand, naming conventions may restrict the allowable renamings. Higher level refactorings may be definable that incorporate domain specific concepts. This implies that a refactoring system should be extensible.

As with any software development tool, it must integrate well with the software development process of the organization that is targeted to use it. The set of refactorings may need to be extended to handle programming styles that, while unusual or non-standard, are nonetheless used in the organization. A refactoring tool must integrate well with other tools that support change management and other software development tasks in an organization [31, 85].

Large software organizations are often resistant to change; changing the names, behavior, interrelationships and responsibilities of objects can have political as well as technical implications. Refactorings need to be prudently applied in these cases.

11.3.6 Some Promising Approaches

There has been research on automating the production of program development environments, and the systems that are produced include the same parts (ie parsers, program transformers, and pretty printers) as a refactory. The most well-known of these systems for generating program development environments are the (Cornell) Synthesizer Generator [95], GANDALF [84] and CENTAUR [22]. Ideally, these systems could be used to generate the software refactory. They have built in facilities for generating parsers that convert source into abstract syntax trees and pretty printers that convert abstract syntax trees into source, and provide other support such as language directed editing and version management. The refactorings themselves are transformations of the parse tree that are much more complex than the simple tree manipulation primitives supported by these systems, but each system has facilities (e.g., action routines in GANDALF and LeLisp functions in CENTAUR) for building upon the system provided primitives.

Early in our research we analyzed CENTAUR, which is one of the most advanced of the program development environments. The language for writing complex tree manipulations was a form of Lisp, so is sufficiently powerful to write any kind of program transformation. However, the overhead of learning to use CENTAUR, and of converting our grammar into a form it could handle, did not seem to be worth the benefits it might provide for our research.

It seems likely that these or other program development environment generators would be suitable for building a production version of the refactory. The time to learn to use such a tool would be amortized over a much longer project life-time, so if it made the rest of the project easier then it would pay off to use it.

11.4 Other Areas for Future Research

There are several other research areas that need further investigation:

1. A more rigorous / complete model of objects and their relationships, with more precise descriptions of the type specifications of classes and a more rigorous proof of behavior preservation, would be a valuable contribution.

2. Program analysis techniques more powerful than data flow could be studied for checking program properties such as class invariants and exclusive components. Class invariants with more complicated properties could be studied (for example, *length(x)*, where x is a member variable representing a list).

3. The set of refactorings could be extended to handle additional features of some object oriented languages, such as multiple inheritance and parameterized types. The approach could also be extended to handle additional features more specific to C++, such as type casts, overloaded function names within a class, and a more powerful handling of pointers.

4. User interface approaches could be studied for assisting a user in making refactoring related design decisions, and reflecting the effects of a refactoring operation.

5. Refactoring and version management are two mechanisms of supporting change that are probably closely related. Refactorings might serve as atomic elements for logging, undo, and recovery. Also, refactoring is made more complicated if operations must be consistently applied to multiple versions of a system.

6. Research is needed into dynamic program analysis (reflection) and refactoring.

   Some program properties cannot always be proven invariant using conventional static analysis techniques, but can more easily be preserved dynamically. For example, conventional static analysis techniques may not correctly determine that a variable can safely be designated as an exclusive component; this limits the use of some refactorings requiring this as a precondition. Dynamically, however, there is a way to ensure that all instances assigned to a particular variable are not already assigned to another component: associate with each instance a tag listing its aggregate(s), and check this tag when assigning an instance as a component. If the instance cannot be assigned as a component, create a replacement or request it from a client/server. Other program properties may also be easier to preserve dynamically than to prove statically. Refactoring an already running system adds complications however; for example, what should be done to instances of classes that were created before the classes were refactored?

11.5 Summary

This thesis has described one type of step in the evolution of a program: refactoring to make the program easier to understand, change and reuse. It has described a set of refactorings, how to check that refactorings are legal, and how to decompose complex refactorings into simpler refactorings. It could serve as the basis for a *software refactory*, which is a program that can refactor another program. While this thesis has focused on refactoring object-oriented application frameworks, it is likely that the refactorings it has described are more generally applicable.

Program evolution and software reuse are important topics that deserve much more attention than they have received thus far. While there are still many problems to solve before tools like a software refactory become a reality, they need to be solved to make software easier to change and reuse. This thesis is a step in their solution.

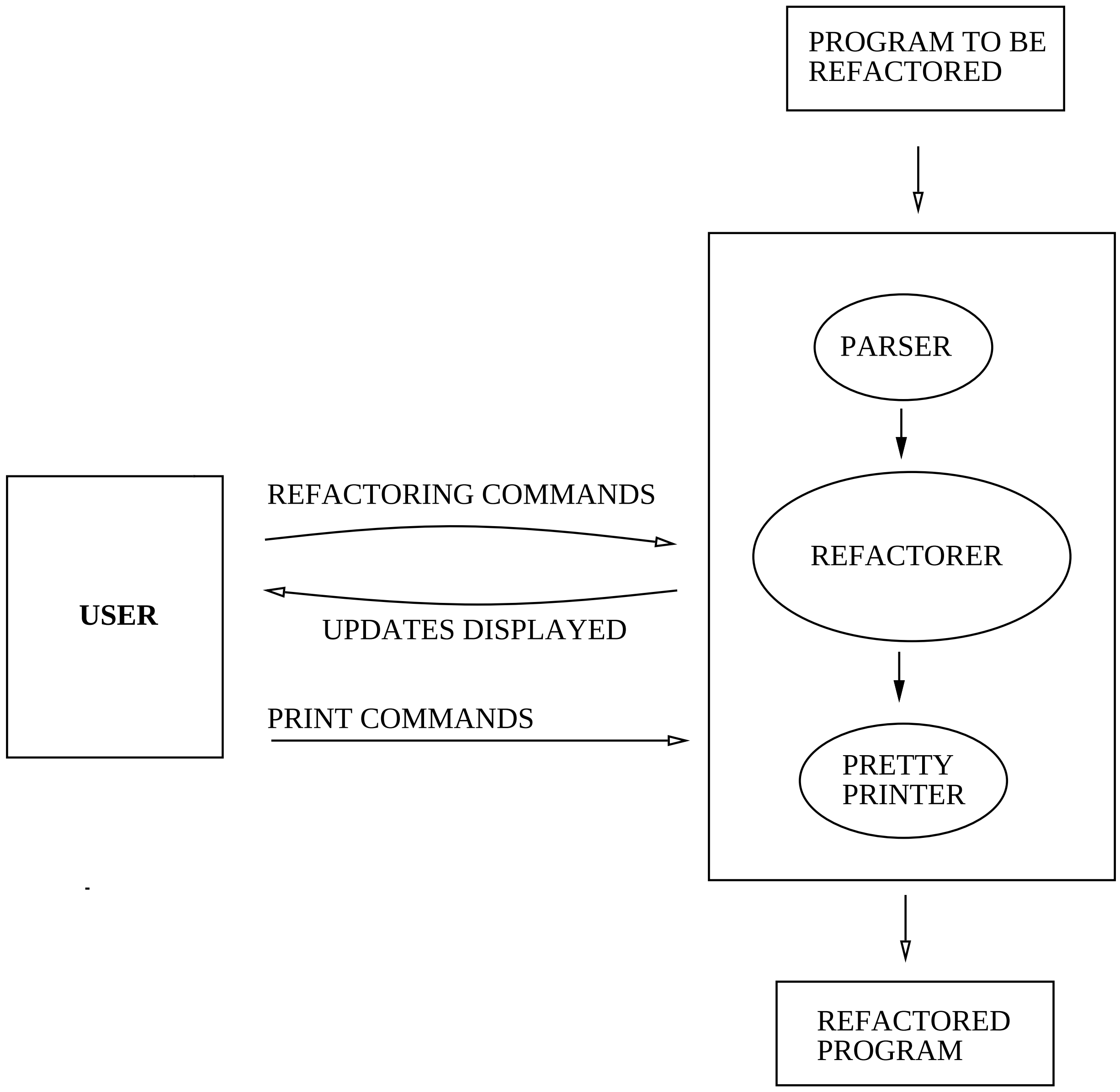


Figure 11.1: Software Refactoring Process

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] E. Allman and M. Stonebraker. Observations on the evolution of a software system. *Computer*, 15(6):27–32, June 1982.

[3] Robert S. Arnold. An introduction to software restructuring. *Tutorial on Software Restructuring (Robert S. Arnold, ed.)*, 1986.

[4] E. Ashcroft and Z. Manna. The translation of "goto" programs in "while" programs. In *Proceedings of the 1971 IFIP Congress*, pages 250–260. North-Holland, 1971.

[5] AT&T. *UNIX System V User Reference Manual.* AT&T, 1984.

[6] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice Hall, 1986.

[7] B. S. Baker. An algorithm for structured programs. *Journal of the ACM*, 24(1):98–120, 1977.

[8] Robert Balzer. A fifteen-year perspective on automatic programming. In *Software Reusability - Volume II: Applications and Experience*, pages 289–311, 1989.

[9] Jay Banerjee and Won Kim. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of the ACM SIGMOD Conference*, 1987.

[10] D. R. Barstow, H. E. Shrobe, and E. Sandewall. *Interactive Programming Environments.* McGraw-Hill, 1984.

[11] V. Basili. *Tutorial on Models and Metrics for Software Management and Engineering.* IEEE Computer Society Press, 1980.

[12] Carol Sue Beckman-Davies. *Finding Program Differences Based on Syntactic Tree Structure.* PhD thesis, University of Illinois at Urbana-Champaign, 1989.

[13] Boris Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, 1983.

[14] G. D. Bergland. A guided tour of program design methodologies. *Computer*, 14(10):18–37, October 1981.

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

[16] Ted Biggerstaff and Charles Richter. Reusability framework, assessment, and directions. *Tutorial: Software Reuse - Emerging Technology (Will Tracz, ed.)*, 1988.

[17] Ted J. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, pages 36–49, July 1989.

[18] Ted J. Biggerstaff and Alan J. Perlis (eds). *Software Reusability - Volume I: Concepts and Models*. Addison-Wesley Publishing Company, Inc., 1989.

[19] Ted J. Biggerstaff and Charles Richter. Reusability framework, assessment, and directions. In *Software Reusability - Volume I: Concepts and Models*, pages 1–19, 1989.

[20] C. Bohm and G. Jacopini. Flow diagrams, turing machines, and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, May 1966.

[21] Grady Booch. *Object-Oriented Design*. Benjamin/Cummings, 1990.

[22] P. Borras and D. Clement. Centaur: the system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 14–24, 1988.

[23] James M. Boyle. Abstract programming and program transformation - an approach to reusing programs. In *Software Reusability - Volume I: Concepts and Models*, pages 361–414, 1989.

[24] Frederick P. Brooks. No silver bullet - essence and accidents of software engineering. *IEEE Computer*, pages 10–19, April 1987.

[25] P. J. Brown. Why does software die? *Infotech State of the Art Report*, 8(7):32–45, 1980.

[26] K. Burns. Using automated techniques to improve the maintainability of existing software. In *DSSD User's Conference/6 - Maintenance*, pages 33–39, 1981.

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

[28] M. H. Burstein. Concept formation by incremental analogical reasoning and debugging. *Machine Learning: An Artificial Intelligence Approach (R.S. Michalski, J. G. Carbonell and T. M. Mitchell, eds)*, 2:351–370, 1986.

[29] R. Canning. Rejuvenate your old systems. *EDP Analyzer*, 22(3):1–16, March 1984.

[30] Eduardo Casais. *Reorganizing an Object System*, pages 161–189. Centre Universitair d'Informatique, Universite de Geneve, 1989.

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

[32] Michael J. Cavaliere. Reusable code at the Hartford Insurance Group. In *Software Reusability - Volume II: Applications and Experience*, pages 131–142, 1989.

[33] Thomas E. Cheatham. Reusability through program transformations. In *Software Reusability - Volume I: Concepts and Models*, pages 321–336, 1989.

[34] Elliot J. Chikofsky. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, January 1990.

[35] Song C. Choi and Walt Scacchi. Extracting and restructuring the design of large systems. *IEEE Software*, pages 66–71, January 1990.

[36] Peter Coad and Ed Yourdon. *OOA - Object-Oriented Analysis.* Prentice-Hall, 1990.

[37] Department of Electrical Engineering Computer Science Division and Computer Science. *UNIX Programmer's Manual, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version.* University of California at Berkeley, August 1983.

[38] Curtis, Krasner, and Iscoe. A field study of the software design process for large systems. Communications of the ACM 31:11, pages 1268–1287, 1988.

[39] N. Dershowitz. Programming by analogy. *Machine Learning: An Artificial Intelligence Approach (R.S. Michalski, J. G. Carbonell and T. M. Mitchell, eds)*, 2:395–424, 1986.

[40] L. Peter Deutsch. Design reuse and frameworks in the Smalltalk-80 system. In *Software Reusability - Volume II: Applications and Experience*, pages 57–72, 1989.

[41] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Co., Reading, MA, 1990.

[42] M. Fagan. Design and code inspection to reduce errors in program development. *IBM Systems Journal*, 15(3):182–212, 1976.

[43] Martin S. Feather. Reuse in the context of a transformation-based methodology. In *Software Reusability - Volume I: Concepts and Models*, pages 337–360, 1989.

[44] Charles N. Fischer and Jr. Richard J. LeBlanc. *Crafting a Compiler.* Benjamin Cummings, 1988.

[45] Gerhard Fischer. Cognitive view of reuse and redesign. *IEEE Software*, 4(4):60–72, 1987.

[46] Brian Foote. *An Object-Oriented Framework for Reflective Meta-Level Architectures.* Ph.D. thesis in preparation, University of Illinois at Urbana-Champaign.

[47] D. Freedman and G. Weinberg. *Handbook of Walkthroughs, Inspections and Technical Reviews (3rd Edition).* Little Brown, 1982.

[48] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, Reading, Massachusetts, 1984.

[49] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[50] J. E. Grass and Y. F. Chen. The C++ Information Abstractor. In *Usenix C++ Conference Proceedings*, pages 265–278, San Francisco, CA, April 1990.

[51] R. Greiner. Learning by understanding analogies. *Artificial Intelligence*, 35:81–125, 1988.

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

[53] Daniel C. Halbert and Patrick D. O'Brien. Using types and inheritance in object-oriented programs. *IEEE Software*, pages 71–79, September 1987.

[54] Patrick A. V. Hall and Geoff R. Dowling. Approximate string matching. *Computing Surveys*, 12(4):381–402, December 1980.

[55] Mehdi T. Harandi and Jim Q. Ning. Knowledge-based program analysis. *IEEE Software*, pages 74–81, January 1990.

[56] M. S. Hecht and J. D. Ullman. Flow graph reducability. *SIAM J. Computing*, 1:188–202, 1972.

[57] Apple Computer Inc. *Lisa Toolkit 3.0.* Apple Computer, 1984.

[58] William A. Jindrich. Foible: A framework for visual programming languages. Master's thesis, University of Illinois at Urbana-Champaign, 1990.

[59] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[60] Ralph E. Johnson, Justin O. Graver, and Lawrence W. Zurawski. TS: An optimizing compiler for Smalltalk. In *Proceedings of OOPSLA '88*, pages 18–26, November 1988. printed as SIGPLAN Notices, 23(11).

[61] W. Lewis Johnson and Martin Feather. Building an evolution transformation library. In *Proceedings of the 12th International Conference on Software Engineering*, pages 238–247, 1990.

[62] Guy L. Steele Jr. *Common LISP: The Language (2rd Edition).* Digital Press, Bedford, MA, 1990.

[63] Simon M. Kaplan. Coed: Conversation-oriented software environments. In *Proceedings of IFIP Conference on Human Facors in Information Systems*, Scharding, Austia., June 1990.

[64] S. T. Kedar-Cabelli. Purpose-directed analogy. In *Proceedings Seventh Annual Conference of the Cognitive Science Society*, Irvine, CA, August 1985.

[65] B. W. Kernighan and P. J. Plauger. *Elements of Programming Style.* McGraw-Hill, 1974.

[66] Won Kim. *Introduction to Object-Oriented Databases.* MIT Press, 1990.

[67] S. R. Kosaraju. Analysis of structured programs. *Journal of Computer and System Sciences*, 9(3):232–255, 1974.

[68] Robert G. Lanergan and Charles A. Grasso. Software engineering with reusable designs and code. In *Software Reusability - Volume II: Applications and Experience*, pages 187–196, 1989.

[69] Karl Lieberherr. *Concepts of Object-Oriented Data Modeling and Programming.* ACM, 1991. Presented as Tutorial No. 16 at OOPSLA '91.

[70] Karl Lieberherr, Walter J. Hursch, and Cun Xiao. Object-extending class transformations (draft). Technical report, College of Computer Science, Northeastern University, 360 Huntington Ave., Boston MA 02115, 1991.

[71] Karl J. Lieberherr and Ian M. Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, September 1989.

[72] Karl J. Lieberherr, Ian M. Holland, and A. Riel. Object-oriented programming: An objective sense of style. In *Proceedings of OOPSLA '88*, pages 323–334, September 1988.

[73] R. C. Linger, H. D. Mills, and R. J. Witt. *Structured Programming: Theory and Practice.* Addison-Wesley, 1979.

[74] Barbara Liskov. Data abstraction and hierarchy. In *Addendum to the Proceedings of OOPSLA '87*, 1987.

[75] M. J. Lyons. Salvaging your software asset (tools based maintanance). In *Proceedings of the National Computer Conference 1981*, pages 337–341. AFIPS Press, 1981.

[76] Peter W. Madany. *An Object-Oriented Framework for File System.* Ph.D. thesis in preparation, University of Illinois at Urbana-Champaign.

[77] Peter W. Madany, Roy H. Campbell, Vincent F. Russo, and Douglas E. Leyens. A Class Hierarchy for Building Stream-Oriented File Systems. In *Proceedings of the 1989 European Conference on Object-Oriented Programming*, Nottingham, UK, July 1989.

[78] J. Martin and C. McClure. *Software Maintenance: The Problem and Its Solution.* Prentice-Hall, 1983.

[79] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

[80] Ware Meyers. Interview with Wilma Osborne. *IEEE Software*, 5(3):104–105, 1988.

[81] Scott Meyers Moises Lejter and Steven P. Reiss. Support for Maintaining Object-Oriented Programs. In *Proceedings of the 1991 Conference on Software Maintenance*, pages 171–178, Sorrento, Italy, October 1991.

[82] H. W. Morgan. Evolution of a software maintenance tool. In *Proceedings of the 2nd National Conference on EDP Software Maintenance*, pages 268–278. US Professional Development Institute, 1984.

[83] James M. Neighbors. Draco: A method for engineering reusable software systems. In *Software Reusability - Volume I: Concepts and Models*, pages 275–294, 1989.

[84] David et al. Notkin. *The Journal of Systems and Software Special Issue: The GANDALF Project.* Elsevier Science Publishing Co., May 1985.

[85] 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 (SOOPPA)*, September 1990.

[86] Tim O'Shea, Kent Beck, Dan Halbert, and Kurt J. Schmucker. Panel on: The learnability of object-oriented programming systems. In *Proceedings of OOPSLA '86*, pages 502–504, November 1986. printed as SIGPLAN Notices, 21(11).

[87] D. L. Parnas. Designing software for ease of extension and contraction. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[88] D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, 5(2):128–138, March 1979.

[89] David L. Parnas and P. C. Clements. A rational design process: How and why to fake it. In *Proceedings of International Joint Conference on Theory and Practice on Software Development (TAPSOFT)*, 1985. Reprinted in *IEEE Transactions on Software Engineering* SE-12:2, February, 1986.

[90] D. L. Parnass, P.C. Clements, and D.M. Weiss. The modular structure of complex systems. In *Proceedings of the 7th International Conference on Software Engineering*, pages 408–417, 1984.

[91] H. Partsch and R. Steinbruggen. Program transformation systems. *Computing Surveys*, 15(3):199–236, September 1983.

[92] D. Jason Penney and Jacob Stein. Class modification in the GemStone object-oriented dbms. In *Proceedings of OOPSLA '87*, 1987.

[93] Edward J. Rak. Two redesign tools for Smalltalk. Master's thesis, University of Illinois at Urbana-Champaign, 1990.

[94] Thomas W. Reps. Incremental evaluation for attribute grammars with unrestricted movement between tree modification. *Acta Informatica*, 25(2):155–178, 1988.

[95] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, 1989.

[96] John R. Rice and Herbert D. Schwetman. Interface issues in a software parts technology. In *Software Reusability - Volume I: Concepts and Models*, pages 125–140, 1989.

[97] Charles Rich and Linda M. Wills. Recognizing a program's design: A graph-parsing approach. *IEEE Software*, pages 82–89, January 1990.

[98] Roxanna Rochat. In search of good Smalltalk programming style. Technical Report CR-86-19, Tektronix, 1986.

[99] Marc J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, December 1975.

[100] James A. Roskind. *YACC compatible C++ grammar (and Related Tools)*. 516 Latania Palm Drive; Indialantic FL 32903 USA. Author can be reached by email at jar@ileaf.com or uunet!leafusa!jar. Software available via ftp from several sites.

[101] James et al Rumbaugh. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[102] Vince Russo, Gary Johnston, and Roy H. Campbell. Process Management in Multiprocessor Operating Systems using Class Hierarchical Design. In *Proceedings of OOPSLA '88*, San Diego, Ca., September 1988.

[103] Vincent Russo and Roy H. Campbell. Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems using Class Hierarchical Design. In *Submitted to OOPSLA '89*, 1989. Also available as University of Illinois Technical Report.

[104] Vincent F. Russo. *An Object-Oriented Operating System.* PhD thesis, University of Illinois at Urbana-Champaign, 1991.

[105] Vincent F. Russo and Simon M. Kaplan. A C++ interpreter for scheme. In *Proceedings of the USENIX C++ Workshop*, pages 95–108, 1988. Also Technical Report No. UIUCDCS–R–88–1461, Department of Computer Science, University of Illinois at Urbana-Champaign.

[106] Barbara Ryder. Paper on incremental data flow ananysis. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, 1990.

[107] Barbara G. Ryder and Marvin C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):231–276, September 1986.

[108] Markku Sakkinen. Comments in the law of Demeter and C++. *SIGPLAN Notices*, pages 38–44, December 1988.

[109] David Sankoff and Joseph B. Kruskal. Macromolecular sequences. In *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison (D. Sankoff and J. Kruskal, eds)*, pages 45–53, 1983.

[110] Walt Scacchi. The USC system factory project. *ACM SIGSOFT Software Engineering Notes*, 14(1):61–82, January 1989.

[111] Stanley M. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6(6):184–186, December 1977.

[112] B. Shneiderman and G. Thomas. An architecture for utomatic relational database system conversion. In *ACM Transactions on Database Systems*, pages 235–257, June 1982.

[113] M. J. Spier. Software malpractice - a distasteful experience. *Software - Practice and Experience*, 6:293–299, 1976.

[114] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Company, 1986.

[115] Kuo-Chung Tai. The tree-to-tree correction problem. *Journal of the ACM*, 26(3):422–433, July 1979.

[116] Walter F. Tichy. Tools for software configuration management. In *Proceedings of the International Workshop on Software Version and Configuration Control (J. F. H. Winkler, ed)*, pages 1–20, 1988.

[117] Will Tracz. *Tutorial: Software Reuse - Emerging Technology.* IEEE Computer Society, 1988.

[118] V. Vyssotsky and P. Wegner. A graph theoretical fortran source language analyzer. Manuscript, AT&T Bell Laboratories, Murray Hill, NJ, 1963.

[119] Robert A. Wagner. Order-n correction for regular languages. *Communications of the ACM*, 17(5):265–268, 1974.

[120] Robert A. Wagner and Michael J. Fisher. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, January 1974.

[121] S. Warren. Map: A tool for understanding software. In *Proceedings of the 6th International Conference on Software Engineering*, pages 28–37. IEEE Computer Society, 1982.

[122] A. I. Wasserman. *Tutorial: Software Development Environments.* IEEE Computer Society, 1981.

[123] G. M. Weinberg. Kill that code! *Infosystems*, pages 48–49, August 1983.

[124] Reinhard Wilhelm. A modified tree-to-tree correction problem. *Information Processing Letters*, 12(3):127–132, June 13 1981.

[125] Rebecca Wirfs-Brock and Brian Wilkerson. Object-oriented design: A responsibility-driven approach. In *Proceedings of OOPSLA '89*, pages 71–75, October 1989.

[126] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software.* Prentice-Hall, 1990.

[127] Rebecca J. Wirfs-Brock and Ralph E. Johnson. A survey of current research in object-oriented design. *Communications of the ACM*, September 1990.

[128] E. Yourdon. *Techniques of Program Structure and Design.* Prentice-Hall, 1975.

[129] Jonathan Zweig and Ralph Johnson. Conduits: A communication abstraction in C++. In *Proceedings of the USENIX C++ Workshop*, pages 191–203, 1990.

Vita

William F. Opdyke attended Drexel University in Philadelphia, Pennsylvania, where he received bachelor of science degrees in computer science and in commerce and engineering sciences, both in 1979. While attending Drexel University he completed computer related cooperative education work assignments at IBM and Sun (Oil) Co. After graduating from Drexel University, he worked for two years in a software quality assurance organization at Sperry Univac.

Dr. Opdyke joined AT&T Bell Laboratories in 1981. He received an MS degree in Computer Sciences from the University of Wisconsin - Madison in 1982, supported under AT&T's graduate study program. Then, at Bell Labs he helped plan the evolution of switched digital telecommunications services, and investigated the application of knowledge-based and related technologies to assist in very large scale software developments. He was approved for the doctoral support program in 1988.

Dr. Opdyke pursued his doctoral research under the supervision of Prof. Ralph E. Johnson. He received his PhD from the University of Illinois at Urbana-Champaign in 1992.

Dr. Opdyke is currently a Member of Technical Staff at AT&T Bell Laboratories. His research interests include object-oriented programming and design, software restructuring and application modeling using object-oriented frameworks.