The Dissertation Committee for Richard Joseph Cardone
Certifies that this is the approved version of the following dissertation:

Language and Compiler Support for Mixin Programming

Committee:

Calvin Lin, Supervisor

Lorenzo Alvisi

Don S. Batory

James C. Browne

Dewayne E. Perry

Language and Compiler Support for Mixin Programming

by

Richard Joseph Cardone, B.Sc., M.S.C.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May, 2002

Dedication

This work is dedicated to Donna and Madeline, without whom it has no meaning.

Acknowledgements

This dissertation is possible because of the best traditions of humanity: The fact that we build institutions of learning, that we revere knowledge and discovery, and that people from all walks of life have the opportunity to express themselves and to contribute to the progress of society. My role is incidental but for the people who have helped me along the way. I take this opportunity to thank those people for their support:

I thank my fellow graduate students, my cohorts, who fought my battles with me and who made my day-to-day work enjoyable and stimulating.

I thank my friends at IBM, especially the management team who took a risk and made this work possible. In particular, I appreciate the efforts of Houtan Aghili, Jacob Ukelson, Kevin McAulliffe, Armando Garcia, Paul Horn, Fran Allen, John Turek and Alfred Spector.

I thank the professors of the Computer Science department, who have created an open and stimulating research environment. I appreciate the interest that my committee has taken in my work and I thank Don Batory, whose pioneering ideas and many hours of conversation inspired my research.

I thank my advisor, Calvin Lin, who showed me what I needed to learn and how to learn it. This research would not exist without Calvin's openness, his confidence in me and, most importantly, his dedication to education. I found a mentor in Calvin; I simply would not have succeeded without him.

I thank my family, whose support and encouragement mean everything. I thank my parents, brother and sister for their dedication and understanding. I thank Madeline for the inspiration she gives me everyday without fail. Lastly, I thank Donna for walking down this path with me, a path I could not walk down alone.

Language and Compiler Support for Mixin Programming

Publication No._____

Richard Joseph Cardone, Ph.D.

The University of Texas at Austin, 2002

Supervisor: Calvin Lin

The need to reduce the cost of software development and maintenance has been a constant and overriding concern since the advent of electronic computing. The difficulty, and therefore the expense, in programming large software applications is due to the complex interactions and interdependencies in application code. These interdependencies increase costs by making code hard to understand, hard to change, and hard to reuse. For over a half century, the need to reduce code complexity has been the driving force behind the trend to program at higher levels of abstraction with increased code modularity.

This dissertation takes a step towards increasing code modularity by showing that *mixin* generic types can be used effectively to build applications from reusable software components. First, we address issues of language definition and integration. We show how mixins can be integrated into a modern programming language to support a methodology of incremental software construction. We

identify novel language and compiler features that make programming with mixins convenient and efficient. Second, we address issues of implementation and evaluation. We implement a critical subset of mixin language support in a compiler. We then use our compiler to show that mixins increase code reuse compared to current technologies; to show that application development and maintenance can be simplified using mixins; and to show that our novel language features simplify mixin programming. In addition, we discuss language implementation issues and define a new design pattern useful in mixin programming.

Table of Contents

List of Tables

List of Figures

CHAPTER 1 INTRODUCTION

1.1 Overview

The 1968 NATO conference [84] on the "software crisis" popularized the term *software engineering*, and ever since that time researchers have been trying to deliver on the promise of systematic and efficient software development that the term implies. In 1979, 43% of the federal project officers surveyed by the General Accounting Office [35] reported that it was fairly or very common for software developed under federal contract to be unusable as delivered. In 1995, a comprehensive survey [106] of over eight thousand public and private software projects reported that more than 30% of the projects were cancelled during development, and of those that were completed, three-quarters were late, over-budget or didn't meet their specifications. The cancelled projects alone were estimated to cost American industry and government $81 billion in 1995.

In addition to the results from studies, well-publicized failures of critical software projects reinforce the perception, both inside and outside the industry, that any large-scale software development is an expensive and risky undertaking. Recent project losses in dollars range from the tens of millions (California DMV's license and registration application), to the hundreds of millions (American Airline's new reservation system, Denver's airport luggage system), to the billions (FAA's air traffic control system) [46,106]. As society's reliance on software becomes more pervasive, the chronic software crisis becomes more acute.

One approach to addressing the software crisis is to make software easier to reuse and, in doing so, to reduce the risk and expense of developing large applications. If existing software can be reapplied in new applications, then the cost of developing and maintaining new code can be avoided. In addition, application quality can increase with code reuse because the code is tested in multiple environments.

In this dissertation, we design an object-oriented programming language, called Java Layers (JL), which extends Java [8] with support for *mixins* [23,117]. Mixins are types whose supertypes are specified parametrically, and they have been shown to increase the reuse potential of code [101,103,126]. Our overall goal is to simplify application development and evolution by increasing the flexibility and reusability of code. We concentrate on *large-scale* applications because of the expense they represent. Large-scale, or simply *large*, applications are complex applications that support variation over time or variation in different execution environments.

Programming with mixins, however, does present a number of challenges. First, mixins differ from conventional classes in that mixins do not have their supertypes specified in their declarations. As a result, mixin hierarchies are not fixed, but instead are assembled on demand. While this under-specification is the source of mixin flexibility, it also means programmers must manage greater variability with less type information. Unless steps are taken to offset the increased complexity of mixin inheritance, application maintainability will suffer. Second,

applications that use mixins are built incrementally in layers, and this can lead to greater runtime overhead due to increased indirection in executable code.

In this dissertation, we show that we can harness the power of mixins and at the same time build efficient applications that are easier to maintain than those built using conventional techniques. We demonstrate that design methodologies, design patterns, and language support tailored for mixin programming provide a practical way to define reusable software components. This ability to reuse code ultimately leads to lower software costs. We describe two evaluations using JL in which increased code reuse made application development and evolution easier. We now describe in more detail the contributions of this dissertation.

1.2 Contributions

The main hypothesis of this dissertation is as follows:

Large-scale software development can be made easier by using mixins and a small number of supplemental language and compiler features.

This hypothesis expresses the idea that mixins realize their true potential when coupled with supporting technology. We now describe the contributions we make in augmenting mixins with supporting technology and in evaluating that new technology.

1. **Feature Identification.**

   We identify supplemental language and compiler features important for mixin programming. We show that mixin programming is enhanced when a small number of supporting language and compiler features are also available. Our contribution includes the definition of novel language features such as the implicit *This* type parameter, constructor propagation, and pattern-based semantic checking. We also define new algorithms for class hierarchy optimization.

2. **Feature Implementation.**

   We show how the identified supplemental features can be integrated into an existing object-oriented language. We build the Java Layers (JL) compiler, which adds support for mixins and our supplemental features to Java. Our contribution is to show that JL's language features integrate well with each other and with Java.

3. **Language Application and Evaluation.**

   We demonstrate the effectiveness of programming large applications using Java Layers. Our contribution is to show that by using mixins and a small number of supplemental capabilities, we improve our ability to develop large-scale software. In our first evaluation, we compare programming in JL with programming using object-oriented frameworks and design patterns, the predominant approach used today to build large applications and software product lines. We evaluate each approach for its flexibility, usability and reusabil-

ity, and we show how JL can be used to avoid problems common to frameworks.

In our second evaluation, we coordinate simultaneous changes to multiple classes using *mixin layers* [103], which are mixins that contain nested types. We show how mixin layers in JL increase code modularity and how this increased modularity can be used to build a software product line from a common code-base.

4. **Other Contributions**

- *We define the Sibling design pattern.* This design pattern can be used with mixins layers to coordinate changes to multiple classes in the same inheritance hierarchy. Our contribution includes defining the Sibling Pattern, demonstrating its usefulness, and providing language support that makes it convenient to use.
- *We design the Class Hierarchy Optimization.* This optimization transforms Java bytecode and, therefore, is applicable to any existing Java application.
- *We describe our mixin implementation.* We document the tradeoffs involved in implementing mixins and in implementing JL's other language features.

1.3 Overview

This dissertation is organized into eight chapters. Chapter 1, Introduction, describes the problem of the high cost of software development, introduces our mixin-based approach to reducing that cost, and summarizes the contributions that our approach makes.

Chapter 2, Motivation, explores the factors that make software development difficult and proposes a mixin-based solution to address these factors. This chapter also describes the challenges of mixin programming and introduces the language and compiler features of Java Layers that address these challenges.

Chapter 3, Related Work, provides the high-level context for our work by describing other approaches to increasing code modularity and to defining mixins. Chapters 4 and 5 describe the work specifically related to each of Java Layers' language and compiler features.

Chapter 4, Language Features, defines Java Layers support for constrained parametric polymorphism. This chapter then describes the design and implementation of four novel language features that support mixin programming.

Chapter 5, Class Hierarchy Optimization, presents the high-level design of the class hierarchy optimization, which is a new optimization that removes the effects of design-time layering from runtime code.

Chapter 6, Compiler Implementation, describes the lessons that we learned from implementing two versions of Java Layers. This chapter also describes the challenges of implementing name mangling in Java.

Chapter 7, Evaluation, describes two evaluations in which we gauge the effectiveness of mixin programming using Java Layers. One evaluation compares mixin programming to programming with object-oriented frameworks. The other evaluation demonstrates how Java Layers can be used to generate cross-platform code libraries.

Chapter 8, Conclusions, summarizes the results of our research and describes possible future work.

CHAPTER 2 MOTIVATION

In this section, we describe in greater detail the problem of software reusability and how current programming technology limits reuse. We then introduce Java Layers (JL), which is our approach to increasing reuse. We talk about the model of software development that JL uses and how mixins provide the technological foundation on which JL is built. Finally, we describe the challenges that mixin programming presents and introduce JL's enhancements to mixins that address these challenges. The full specification of JL's language and compiler enhancements is given in Chapter 4, Language Features, and Chapter 5, Class Hierarchy Optimization.

2.1 The Software Challenge

Large-scale software development is difficult because applications are implemented in dynamic environments that are characterized by two important types of variation. First, *variation over time* results from the nearly constant flow of new requirements and new demands that are placed on applications. This pressure to change leads to the implementation of new application *features*, which we define as any characteristic or capability that an application supports. As features are added, removed or modified, unanticipated interactions and co-dependencies between feature implementations decrease the overall modularity of the software. Over time, these incremental changes tend to degrade the quality of an applica-

tion. In many applications, design decays to the point where either an expensive redesign is required or the application becomes so resistant to change that it must simply be discarded.

It is hard to over-emphasize the importance of planning for variation over time, or for the *maintainability*, of an application. Studies indicate that development organizations spend 60% to 80% of their budget on software maintenance [93]. Anecdotal evidence supports this assessment. For example, the Windows NT's code base grew at an annual rate of 33% in the four years after its release, which tripled its initial size to surpass 30 million lines of code by 1997 [83]. This level of maintenance activity occurs in software that supports the changing needs of many users.

The second type of variation that large applications need to support is *variation in execution environments*. This type of variation requires that different versions of an application support different users, hardware platforms, or market segments at the same time. The need to provide different features in different environments leads to the development of families of applications or *software product-lines* [22,50]. The challenge here is to reduce the cost of building and maintaining product lines by maximizing the reuse of design and code, and to do this without sacrificing performance or maintainability. This goal is difficult to achieve because the requirements of multiple execution environments must be considered simultaneously.

A particular concern that arises when developing software product-lines is the *feature combinatorics* problem [16]. Given a domain with ***n*** optional features,

the feature combinatorics problem occurs when all valid feature combinations must be predefined or in some way materialized in advance. In the worst case, 2^n concrete programs would have to be instantiated. To efficiently produce software product-lines, we must be able to easily customize and reconfigure applications for specific uses. The goal, therefore, is to maximize the flexibility and reuse of existing code while avoiding the maintenance problems that a combinatoric explosion of code would cause.

The need to support variation over time and variation in execution environments makes software development complex. One way to manage this complexity is to avoid, as much as possible, the creation of new code by reusing code that already exists. In general, less code means less maintenance; in software product-lines, the ability to avoid duplicated effort determines the viability of the application. We now describe the basic characteristics of reusable code and how reuse is limited by current technology.

2.2 Limitations on Reuse

The goal of reuse is to build large applications from *reusable software components*. The ability to reuse code depends on two properties: *modularity* and *easy composition*. Modularity allows us to separate concerns [91], which makes code easier to understand, maintain, and treat as a unit. Easy composition allows us to combine the capabilities of different code modules into different applications.

If application features can be implemented as reusable software components, then we could build applications by *mixing and matching* features. In the ideal, we would build custom applications by selecting and composing the features we need without writing any new code. Of course, feature code still needs to be written at some point; but once written, it can be used in more than one application. Our ability to program at this higher level of abstraction depends on feature implementations that support both modularity and composability. We now describe how support for these two properties falls short in current programming technologies.

The first deficiency is that current programming technologies cannot completely encapsulate feature implementations. In object-oriented languages like Java, for example, the basic unit of encapsulation and reuse is the class. Once the organization of classes in a program is fixed, it is always possible to define new features whose implementations *crosscut* the existing set of classes [66,118]. For example, it is common for features that define global program properties to affect the code in multiple classes. Such global properties include security, thread safety, fault tolerance, and performance constraints. Generally speaking, object-oriented programs consist of sets of collaborating classes [44], and changes to one class often require coordinated changes to other classes.

To illustrate the current limits on encapsulation, we use an example from our graphical user interface (GUI) evaluation, Fidget (§7.2). Fidget implements GUI widgets, such *Window*, *Button*, and *TextField*, in their own class. In Fidget, support for color displays is an optional crosscutting feature that can be applied to

a library of widgets. Using standard object-oriented techniques, however, color support breaks encapsulation and limits reuse. There are two reasons for this. First, color support cannot simply be inherited from a superclass because individual widgets, implemented in their own classes, provide specialized color processing. Thus, the code implementing color support is *scattered* [51] among multiple widget classes, making the code difficult to reuse and difficult to remove. Second, widget classes commingle code for color support with that of other features. This *tangling* [51] of feature code in a class makes the class more complex, more interdependent with other classes and, ultimately, more difficult to reuse.

The second deficiency is that current programming technologies are limited in their ability to compose features. For example, Java's support for composition depends primarily on single inheritance and subtype polymorphism, which do not scale well when there are a large number of optional features.

To illustrate this scalability problem, we use another example from Fidget. Consider the possible features that a *TextField* widget might have: the ability to query or change the font; to echo input; to choose the echo character set; to allow for selection, cut, paste, drag and drop; to support resizing; and to support different styles of event handling—we could list more. By encapsulating each optional feature in its own class, we can build a custom *TextField* widget by creating a class hierarchy with a base class and the selected feature classes in linear order. The result is a fixed class hierarchy that supports the chosen text field features. Different combinations of features, however, require different hierarchies. In some cases, these new hierarchies would require an existing feature class to have

a different superclass, which leads to code replication. This code replication, which quickly becomes unmanageable as the number of different feature combinations increases, is an example of the feature combinatorics problem described in the last section.

To alleviate the limitations in modularity and composability in current programming languages, we propose to increase the expressiveness of languages to better support reuse. We now describe concepts and technology behind our language proposal.

2.3 The Java Layers Solution

In the last section, we described how code reuse is limited in current programming languages. To address this problem, we designed Java Layers (JL) as an extension of Java [8] that enhances support for reuse. We chose to work with Java because of its widespread use and because of the good software engineering characteristics that it already embodies. These characteristics include simplicity in design, relatively pure object-oriented semantics, a high-level memory model, integrated exception handling, and static type checking,

This section describes the programming model used by JL and the mixin technology used to implement that model. We begin with the conceptual framework that allows JL programmers to systematically plan for reuse.

2.3.1 THE GENVOCA MODEL

The GenVoca model of software development, which was introduced by Batory and O'Malley [15], provides the conceptual foundation for programming

in Java Layers. The GenVoca model consists of software components called *layers*, compositions of layers called *type equations*, and a programming methodology that emphasizes *stepwise program refinement*.

A GenVoca layer encapsulates the complete implementation of a single application feature. Layers can contain code that crosscuts the modules or constructs of a programming language. For example, in object-oriented languages, layers can contain code that affects multiple classes or multiple methods. Layers *export* an interface and *import* zero or more interfaces, where an interface consists of all externally visible characteristics of a layer. The interface exported by a layer defines its *realm*, and layers that export the same interface are members of the same realm.

Layers are composed in type equations, which match the exported interfaces of actual layer parameters to the formal interface parameters of an importing layer. Layers that export the same interface can be interchanged with one another, though their implementations are different.

The example in Figure 1 is taken from the original GenVoca paper [15]; it shows three layers and a type equation that relates them. We assume that each layer imports the interface it needs, since realms are not shown. The arrows represent call relationships between the layer nodes. There is an externally visible layer (*layer1*) on which calls are invoked. Generally speaking, a layer performs its feature's processing and then passes control to the next layer. This call forwarding ends in a terminus layer (*layer3*). Layers can be thought of as virtual

machines that process requests at their level in the hierarchy and then pass the requests to the next layer for further processing.

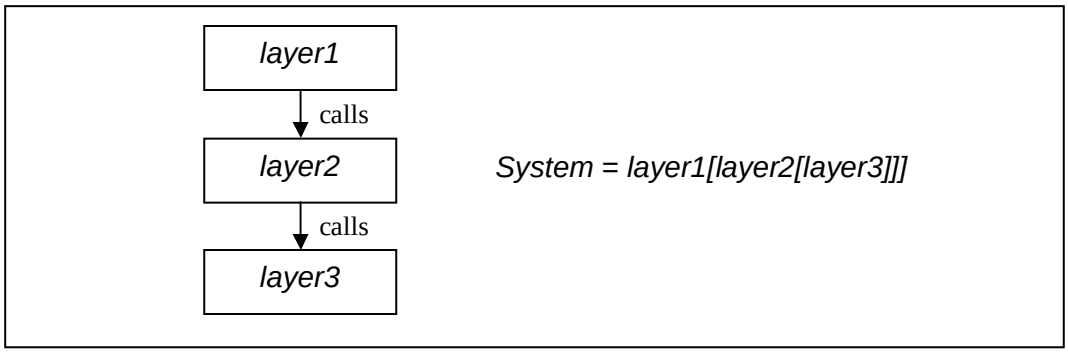


Figure 1 - Stacking GenVoca Layers

The key GenVoca idea is that software systems can be created by layering reusable, parameterized, software components. In Figure 1, *System* is the program *generated* from the composition of three features, where each feature is implemented in its own layer. This methodology of building programs incrementally in layers is called *stepwise* or *layered refinement*. Layers are sometimes called *large-scale refinements* because they implement crosscutting features that refine program behavior.

GenVoca supports the reuse properties of modularity and composability described in §2.2. In GenVoca, applications can be built by mixing and matching features because layers encapsulate feature implementations in reusable and composable components. GenVoca avoids the feature combinatorics problem (§2.1) because new programs are only generated when a new combination of features is required.

The GenVoca model does not specify an implementation technology, but object-oriented languages that support parametric polymorphism [29] already provide a number of key capabilities needed to implement the model. First, these languages provide classes as encapsulation constructs that can model the encapsulation of GenVoca layers. Second, parameterized classes can model the composition of GenVoca layers. To effectively model layer composition, however, parameterized classes cannot be fixed in a single hierarchy. We now describe how *mixin classes* satisfy this need for compositional flexibility and how they provide the foundation for JL's implementation.

2.3.2 MIXINS

The term *mixin* was first used to describe a style of LISP programming that combines classes using multiple inheritance [62,79]. Since then, however, mixins have been more commonly defined as types whose supertypes are declared parametrically [23,117], and it is in this sense that we use the term. JL supports mixins and other generic types by implementing *parametric polymorphism* [29], which allows types to be declared as parameters to code. Mixins are useful because they allow multiple classes to be specialized in the same manner, with the specializing code residing in a single reusable class. In addition, mixins provide the compositional flexibility needed to implement GenVoca layers.

To understand the benefits of using mixins, we consider an example. Suppose we wanted to extend three unrelated Java classes—`Car`, `Box` and `House`—to have a "locked" state by adding two methods, `lock()` and `unlock()`. Without mixins, we would define subclasses of `Car`, `Box`, and

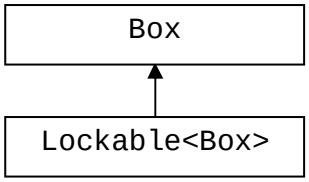
House that each extended their respective superclasses with the lock() and unlock() methods. This approach results in replicating the lock code in three places. Using mixins, however, we would instead write a single class called Lockable that could extend any superclass, and we would instantiate the Lockable class with Car, Box, and House. This approach results in only one definition of the lock code. In JL, the Lockable mixin would be defined as follows:

```
class Lockable<T> extends T {
 private boolean _locked;
 public lock(){_locked = true;}
 public unlock(){_locked = false;} }
```

The above class is *parametric* because it declares *type parameter* T. JL's parametric types are similar in syntax and semantics to C++ [110] template classes. When Lockable<T> is compiled, T is not bound. To use Lockable<T>, T must be bound to a type to create an *instantiation* of the parametric class. Each distinct binding of T defines a new *instantiated type*, which can then be used like a conventional Java type.

What makes Lockable<T> a mixin, however, is that its instantiated types inherit from the types bound to T. Mixins are distinguished from other parametric types because the supertypes of mixins are specified using type parameters. Thus, a mixin's supertypes are not known at *compile-time*, but instead are specified at *instantiation-time*.

Mixin instantiations generate new class hierarchies. For example, Lockable<Box> generates the following hierarchy:



In the above mixin-generated hierarchy, `Lockable<Box>` is the *leaf* class and `Box` is the *root* class. Mixins can also generate hierarchies with more than two classes. For example, suppose we define a `Colorable` mixin to manage a physical object's color and we define an `Ownable` mixin to manage ownership properties. We can now create a variety of physical objects; these objects can support various combinations of features and can generate hierarchies that contain several classes, as the following instantiations illustrate:

```
Colorable<Ownable<Car>>
Colorable<Lockable<Box>>
Lockable<Ownable<Colorable<House>>>
```

We can think of each of the above *mixin compositions* as starting with the capabilities of some base class (`Car`, `Box` or `House`) and refining those capabilities with the addition of each new feature. In the end, we produce a customized type that supports all the required features. When used in this way, mixins can be thought of as *type fragments* because each mixin provides only some of the capabilities needed for the complete type.

Mixin classes derive their flexibility from their ability to be easily *reparented*. This ability to inherit from different parent classes allows mixins to specialize different classes with the same feature. Reparenting is similar to adding

another superclass to a class in languages that support multiple inheritance, but mixins avoid the pitfalls of multiple inheritance [117].

The ability of mixin classes to extend different superclasses gives them the compositional flexibility necessary to implement GenVoca layers. The encapsulation capabilities of mixins, however, are limited in that they can only affect the code of one class. Smaragdakis and Batory address this limitation by developing the idea of *mixin layers*, which we now discuss.

2.3.3 MIXIN LAYERS

Mixin layers [103] are mixins that contain nested types. This nested structure can implement features that crosscut multiple classes. To see how this works, we revisit the Fidget evaluation that we introduced in §2.2. The code below shows a simplified version of the basic Fidget class and one of its mixin layers. `BaseFidget` contains the basic implementation of all widgets, two of which are shown (`Button` and `CheckBox`). `LightWeightFidget` implements display support, which is a crosscutting feature that affects all widget classes. `LightWeightFidget` is a mixin layer that contains nested mixin classes.

```
class BaseFidget {
  public class Button {…}
  public class CheckBox {…} …}

class LightWeightFidget<T> extends T {
  public class Button extends T.Button {…}
  public class CheckBox extends T.CheckBox {…} …}
```

Figure 2 below shows the three hierarchies that are generated when `LightWeightFidget` is instantiated with `BaseFidget`. Note that each of the

nested classes in `BaseFidget` is extended by its corresponding class in `LightWeightFidget`. In this way, multiple widget classes are specialized with display support simultaneously using the `LightWeightFidget` mixin layer.

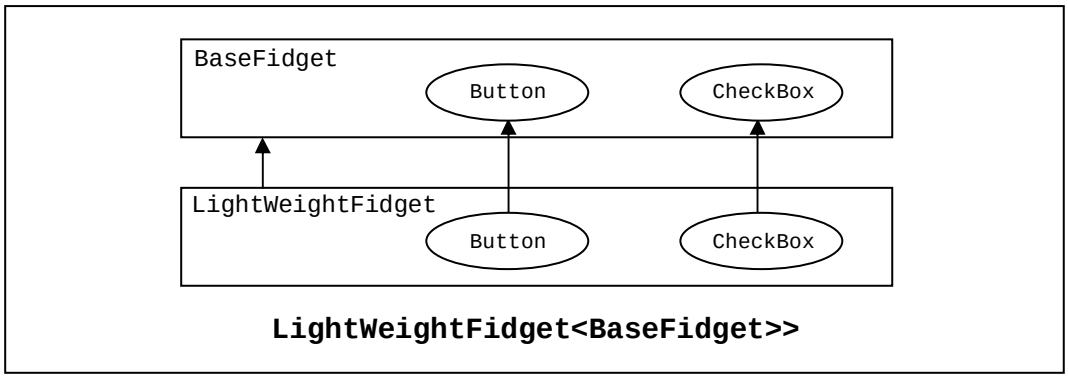


Figure 2 - Mixin Layer Instantiation

The `LightWeightFidget` mixin layer encapsulates feature code that is scattered across multiple classes, which demonstrates that mixin layers can implement large-scale refinements. With mixin layers, we now have both the modularity and the composability necessary to make application code more reusable. In JL, we use mixins and mixin layers to implement the GenVoca model and its methodology of stepwise program refinement.

The reuse benefits of mixin layers have been documented [101,103]. The contribution of our work is to show that mixins and mixin layers can be integrated into a programming language so that their benefits can be more fully realized. In the next section, we describe the challenges of mixin programming and the novel ways that we address these challenges in JL.

2.4 Challenges of Mixin Programming

In this section, we describe the challenges that motivate our work. Our goal is to use mixins to produce more maintainable and more reusable code. In the Introduction, we noted that mixins have the potential to make programming more complex and to produce more runtime overhead. In the subsections below, we explore these challenges in depth and we introduce mechanisms designed to address them. In Chapters 4 and 5, we discuss the design and implementation of these mechanisms in detail.

2.4.1 INITIALIZATION

Superclass initialization is not straightforward in mixin classes because the superclass of a mixin is not known when the mixin is defined [126]. This under-specification is the source of mixin flexibility, but it can also be the reason why mixin compositions fail to compile. In this section, we describe the problem of mixin initialization and how it is exacerbated by stepwise program refinement. We discuss common static approaches to the initialization problem and introduce JL's dynamic approach, which is a flexible new way to automatically generate constructors.

Figure 3 shows a simple case in which an invalid constructor invocation causes a mixin composition to fail. Since mixin `M`'s type parameter is unconstrained, any class can be specified as `M`'s superclass. In particular, instantiating `M<A>` causes a linkage failure because `M` implicitly calls the no-argument constructor for class `A`, which does not exist.

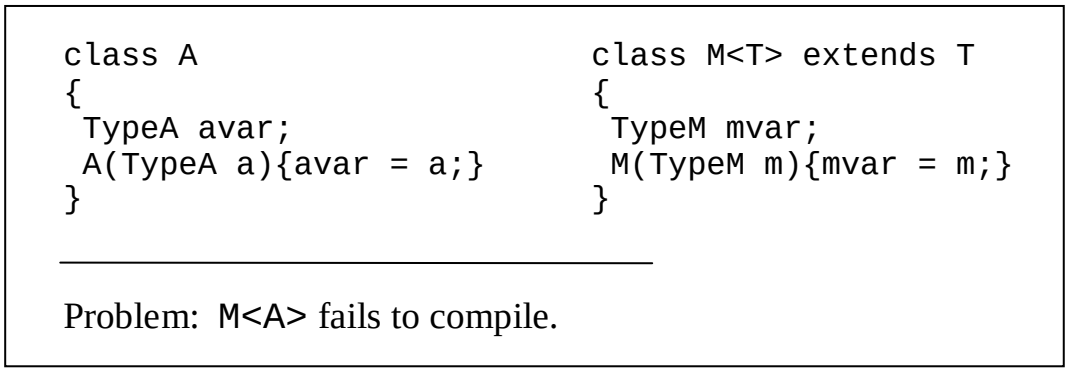


Figure 3 - Constructor Mismatch during Instantiation

The above initialization of `M<A>` would succeed if a call to `A`'s constructor were inserted into `M`'s constructor. Unfortunately, this solution substitutes one constructor dependency for another: All of `M`'s superclasses would now have to support a constructor with a single `TypeA` argument. We characterize such approaches to mixin initialization as *static* approaches because they assume fixed-signature superclass constructors. Static approaches are common in practice, but they limit mixin reuse by introducing new compositional dependencies.

Static approaches range from the minimalist to the all-inclusive. The minimalist approach requires that mixin superclasses implement a small number of constructors that take well-known arguments. This approach works well when the same arguments are used in most, if not all, instantiations. If some class in an instantiation needs auxiliary initialization data, then a user-defined initialization method can be used to pass that data after object construction. This custom initialization protocol for exceptional cases is enforced by programming convention.

On the other hand, the all-inclusive static approach requires that mixin superclasses implement a single constructor that takes any argument used in any instantiation. Each class in the instantiation uses the arguments it needs and ignores the rest. Inevitably, null references and references to dummy arguments are used to reduce overhead in specific instantiations. Such attempts to increase efficiency complicate code because programmers have to test arguments before using them.

There are a number of possible variations on the minimal and all-inclusive static approaches to superclass initialization, including variations that mix the two. Static approaches are acceptable in applications where initialization arguments do not vary much or in applications that do not contain many mixins. In large applications or in applications that exhibit significant variation in initialization data, however, a more dynamic approach is needed.

The need for a more dynamic approach is reinforced when one considers how mixins are used. In §2.3, we described how stepwise program refinement encourages the use of small, single-featured, mixin classes to incrementally build applications. The finer the decomposition is, the greater the number of mixins in an application. Most of these mixins do not need explicit initialization, but they often have to define constructors just to forward initialization parameters to their superclasses. These *forwarding constructors* increase the housekeeping chores for programmers and become more burdensome as the number of mixins increase.

In §4.4, we describe *constructor propagation*, which is JL's *dynamic* approach to superclass initialization. Dynamic approaches do not assume fixed-signature superclass constructors. In JL, mixin reuse is increased because mixin

constructors can adapt to their superclass constructors at instantiation-time. In addition, constructor propagation makes stepwise program refinement more convenient by automatically generating forwarding constructors.

2.4.2 SELF-TYPE REFERENCES

Mixin programming alters the way we think about the *self-type references* in object-oriented code. The *self-type* of a code fragment is the statically-known type that contains the code. References to self-types are ubiquitous in object-oriented code. For example, if method *m* in class *C* contains the expression *new C()*, then *m* contains a self-type reference.

In Java, self-type references have two characteristics that make their use convenient and effective. First, self-type references always refer to the declaring type that contains the reference, which implies that the self-type's name is known when the code is being written. Second, all the capabilities of a type are available through its self-type references.

Mixins, however, define type fragments. Mixins are used to implement a single application feature that does not stand on its own, but must be combined with other code to deliver its function. When a mixin is defined, the mixin name is known, but the name of the type that ultimately combines the mixin's function with the function of other code is not known. Also, if a mixin is used as the self-type for the code it contains, then only the capabilities of that mixin are available through self-type references. Using mixins as self-types means that only some of the capabilities specified by a mixin composition are accessible.

In mixin programming, the leaf type generated by a mixin composition is the full-featured type. This leaf type is sometimes referred to as the *ultimate type* or the *most derived type* generated by a composition. Each distinct mixin composition generates its own ultimate type.

In §4.3, we introduce JL's implicit **This** type parameter, which allows us to express the most derived class of a mixin composition. **This** is implicitly defined in parametric types and can be used like other type parameters. In §4.3, we also describe how **This** is useful in defining recursive types.

2.4.3 CONTROLLING COMPOSITION

Mixins provide a powerful way to compose software, but supporting mechanisms are needed to manage mixin composition. This section describes the problems of unrestricted mixin composition and introduces three JL language features that increase programmer control over composition.

Mixins offer great flexibility by deferring the specification of parent/child relationships from definition time to composition time. This flexibility, however, increases the likelihood that syntactically correct compositions generate programs with undesirable, unpredictable or invalid behavior. Undesirable program behavior occurs when mixins in a composition reduce performance or increase program size without adding function. Unpredictable program behavior occurs when mixins interfere with each other's execution in subtle and hard-to-detect ways. Invalid program behavior occurs when mixins cause compilation or runtime failures.

We categorize the challenges in controlling mixin composition as (1) those involving type parameter bindings, (2) those involving combinations of mixins,

and (3) those involving nested class structure. For each of these three categories, we use examples to illustrate the challenge and then introduce the JL language feature that addresses the challenge.

We use the network transport code shown in Figure 4 to illustrate the challenges of mixin composition. The `TCP` class provides data transport using TCP; the `Secure` mixin adds data privacy; and the `KeepAlive` mixin automatically exchanges liveness notifications between communicating peers.

```
class TCP
{
 public void send(byte[] out){…}
 public void recv(byte[] in){…}
 public void disconnect(){…}
}

class Secure<T> extends T
{
 public void send(byte[] out)
  {byte[] buf = encrypt(out); super.send(buf);}
 public void recv(byte[] in){… super.recv(buf); …}
 public void disconnect(){… super.disconnect(); …}
}

class KeepAlive<T> extends T
{
 protected void keepAlive(){…}
 public void disconnect()(){… super.disconnect(); …}
}
```

Figure 4 - Unrestricted Mixin Composition

The first kind of composition challenge involves invalid type parameter bindings. The instantiation `Secure<TCP>` compiles because all superclass references in `Secure` can be resolved in `TCP`. On the other hand, the instantiation `Secure<java.util.HashMap>` fails to compile because superclass references in

`Secure` cannot be resolved in `HashMap`. The source of the problem is that `HashMap` doesn't support the interface expected by `Secure`. This problem is similar to the problem of unresolved constructor references described in §2.4.1, but here the unresolved references are to superclass members.

Constrained parametric polymorphism [29,109] is a common solution to the problem of invalid type parameter bindings. This solution allows programmers to explicitly restrict the types that get bound to a type parameter, which means that invalid bindings can be detected early using static type checking. Constrained parametric polymorphism increases code *comprehensibility* by allowing explicit type restrictions on type parameters; it increases language *usability* by allowing compilers to report meaningful error messages; and it improves code *robustness* by more closely integrating type parameters into the type system. JL's implementation of constrained parametric polymorphism is described in §4.1.

The second kind of composition challenge involves managing the numerous ways that mixins can be combined. For example, the order in which mixins are composed can affect whether compilation succeeds and whether the generated program behaves as expected. Referring again to Figure 4, both `KeepAlive<Secure<TCP>>` and `Secure<KeepAlive<TCP>>` generate code that supports a secure TCP transport with automatic keep-alive. Liveness notifications, however, are transmitted encrypted in the first instantiation and in the clear in the second instantiation. In our example, both orderings could be supported, but this is not true in general. Indeed, sometimes a mixin can only be used if another mixin appears either before or after it in a composition.

In addition to order, the presence, absence, or number of times a mixin appears in a composition also affects program behavior. For example, the meaning of `Secure<Secure<TCP>>` and whether `Secure` can be validly used twice in the same composition is implementation dependant. In general, mixins can be implemented in ways that either require or disallow the use of other mixins in the same composition.

Mixin programming is scalable only if composition rules involving the presence, absence, cardinality and ordering of mixins can be automated. In §2.1, we saw that as the number of optional features increases, the number of possible feature combinations increases exponentially. Programmers cannot be expected to manage this level of complexity, even with good documentation. In §4.5, we describe a way to manually encode *semantic checking* rules into class definitions so that a compiler can detect invalid feature combinations automatically. JL's semantic checking goes beyond the syntactic capabilities of type checking and is an alternative to previous GenVoca approaches [11,13].

The third kind of composition challenge involves the propagation of nested class structure when mixin layers are used. In §2.3.3, we described how a mixin layer can encapsulate the complete implementation of a feature whose code resides in multiple classes. When an application is composed using mixin layers, each layer expects its superclass to have a certain nested structure. Conversely, each mixin layer needs to present a certain nested structure to its subclasses.

Figure 5 shows a configuration of Fidget classes, which are part of our evaluation in building families of graphical user interface libraries (§7.2). The

figure illustrates a composition of three layers that is generated by the instantiation shown. Two of Fidget's nested classes, `Checkbox` and `Button`, are shown in each of the layers.

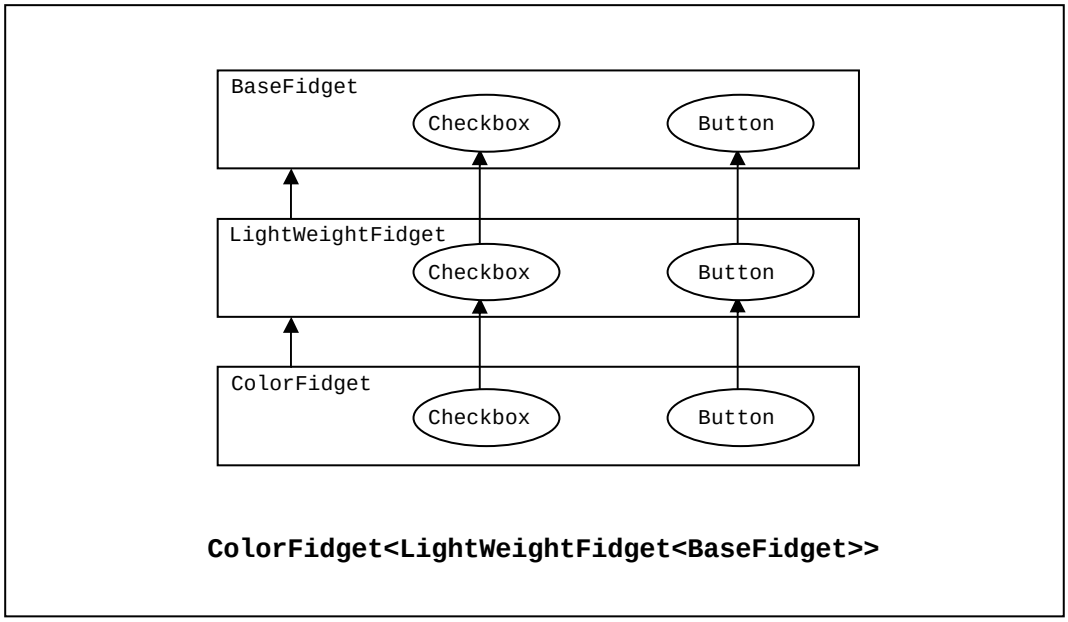


Figure 5 - Fidget Example of Mixin Layer Usage

The regularity of inheritance relationships shown in Figure 5 promotes the use of stepwise program refinement (§2.3.1). First, each mixin layer expects its superclass to contain `Checkbox` and `Button`, and these nested classes are expected to be subclasses of `Checkbox` and `Button` in `BaseFidget`. Second, each mixin is expected to provide these nested classes to its subclasses. Mixin layers that meet these two conditions can be composed with one another to build complete libraries one feature at a time. In §4.2, we describe how JL enforces these two conditions through its implementation of *deep conformance*. JL's novel

design is the first implementation of the deep conformance properties originally defined by Smaragdakis [101].

In summary, we address the challenges of controlling mixin composition in JL by implementing three language features: constrained parametric polymorphism, semantic checking, and deep conformance. These language features allow programmers to restrict how mixins are combined by controlling the following elements of composition:

- The types that can be bound to a type parameter
- The ordering of mixins
- The number of times a mixin can be used
- The presence or absence of a mixin
- The nested structure of actual type parameters
- The nested structure of generated types

2.4.4 RUNTIME EFFICIENCY

In this section, we discuss performance in layered applications. We describe how deeply layered code can reduce performance, and we summarize the results of micro-benchmarks that reinforce that idea. We then introduce an optimization that reduces the overhead of layering.

Stepwise program refinement (§2.3.1) uses mixins to encapsulate the implementations of fine-grained application features. Applications are built by combining mixins until all required features are included. This programming

methodology can lead to applications that consist of many small classes layered in deep hierarchies. In Java, two questions about performance are raised when code is organized this way.

First, the use of many small classes in a program can increase load time. In addition, the management of many classes at runtime increases the memory used by a Java Virtual Machine (JVM).

Second, stepwise refinement often introduces the runtime overhead of extra method dispatches. In mixin-generated hierarchies, methods with the same signature are often daisy-chained together so that each feature's code executes when a method is called. For example, when method *m* in the leaf class of a mixin-generated hierarchy is called, *m* typically executes the leaf class's feature code and then calls *m* in the superclass. This second method invocation executes the code for the superclass's feature. This succession of method calls continues up the hierarchy until some terminus class is reached. By contrast, conventional implementations intermix the code for multiple features in the same method, which avoids the cost of a method call when different features execute.

To quantify the effect of design-time layering on runtime performance, we performed two sets of micro-benchmarks. A description of these benchmarks and a discussion of their results are in Appendix A – Layered Code Micro-Benchmarks on page 255. These benchmarks reinforce the concerns described above and their results can be summarized as follows:

1. Eliminating long chains of method calls through inlining can lead to significant speedups.
2. Many small files take longer to load than fewer larger files, assuming the total number of bytes loaded remains constant.

In addition, a recent performance analysis of mixins in C++ provides more evidence that inlining can significantly improve the performance of mixin-generated code. Berger, Zorn and McKinley [17] build customizable memory allocators using mixins and show that even though mixins introduce many layers of abstraction, aggressive method inlining results in mixin-generated code that performs at least as well as, and sometimes better than, equivalent conventionally written code.

The *class hierarchy optimization* described in Chapter 5 addresses both of the above performance concerns. In our optimization, calls to superclass methods with the same signature are aggressively inlined and the whole class hierarchy is collapsed into as few classes as possible. As long as certain constraints are satisfied, our optimization can be applied to the bytecode of arbitrary Java class hierarchies.

CHAPTER 3 RELATED WORK

In this section, we discuss two areas of related work that provide the high-level context for our research. The first area describes other approaches to increasing the modularity of code by relating our work to prominent non-mixin technologies that also focus on reusability. The second area describes recent theoretical work on mixin types and relates our work to these approaches to defining mixins.

This chapter is necessarily high-level because we have not yet described Java Layers in detail. We do, however, present detailed discussions of related work in Chapters 4 and 5. This two-level organization allows us to discuss work related to JL features in the context of those features. Thus, the discussion of each JL language or compiler feature includes its own Related Work section. The table below provides an index to these feature-specific Related Work sections.

Feature	Related Work Section	Related Work Page
Constrained Parametric Polymorphism	§4.1.4	48
Deep Conformance	§4.2.5	71
The Implicit *This* Type Parameter	§4.3.5	93
Constructor Propagation	§4.4.4	103
Semantic Checking	§4.5.4	121
Class Hierarchy Optimization	§5.9	185

Table 1 - Index of Related Work Sections

3.1 Increasing Modularity

Programming languages have steadily evolved to support higher levels of abstraction and more powerful encapsulation capabilities. Beginning with machine and assembly code, and progressing through structured code, abstract data types and object-orientation, programming languages have increased their ability to separate design concerns in code. This increased modularity has led to greater reuse and other engineering benefits. Mixins continue the trend towards greater modularity by making object-oriented inheritance more flexible. In §2.3, we described how Java Layers uses mixins and stepwise program refinement to build applications from reusable software components.

In this section, we describe two other approaches to increasing reuse. First, Aspect-Oriented Programming provides a general model for specifying

crosscutting components. Second, Multi-Dimensional Separation of Concerns provides an even more general model that addresses the separation of concerns in all software-related artifacts, including requirements, design and code. We conclude our discussion of modularity by relating Java Layers to previous GenVoca research.

3.1.1 ASPECT-ORIENTED PROGRAMMING

The Aspect-Oriented Programming (AOP) [66] model defines two basic concepts. The first concept is that of a *base language*, which is the primary abstraction and composition mechanism that is used to organize code into *implementation units*. Different base languages have different implementation units. For example, procedural, functional, and object-oriented languages respectively define procedures, functions, and objects as implementation units. Using base languages, application function is encapsulated in implementation units.

AOP's second basic concept is the *aspect*. Aspects implement new application features by transforming base language code. Aspects can modify code in multiple implementation units, which allows aspects to encapsulate crosscutting features. The modifications specified in aspects take place at predefined *join points*. Join points can be defined generally in terms of base language constructs, such as class or method declarations, which allow aspects to transform any application written in that language [65]. Alternatively, join points can be defined in terms of a specific application, which allow aspects to make application-specific transformations and optimizations [77,128]. For instance, application-specific

join points can be defined in terms of specific control flow patterns or field reference patterns.

AOP provides a general model for thinking about crosscutting features, and the term *aspect* has become synonymous with crosscutting feature encapsulation. In AOP, *aspect languages* are implemented in compilers called *weavers*, which combine aspect code and base language code in customized ways. Indeed, Java Layers can be thought of as an aspect language in which mixin layers are the aspects and inheritance is used to weave code.

AspectJ [65] is an application-independent AOP extension to Java in which aspects implement features that crosscut class boundaries. The base language is Java; the methods and classes of Java serve as AspectJ's join points. Aspects can add new methods to existing classes and they can weave code before or after the execution of methods.

Mixin layers have many of the same capabilities as AspectJ's aspects. Like aspects, mixin layers can define new methods in classes. Also, by using method overriding and explicit calls to **super**, mixin layers can specify code that executes before or after existing methods. AspectJ's aspects, however, can refine the behavior of an arbitrary group of classes, while mixin layers can only refine the classes nested in their superclasses. Thus, aspects are more expressive and can address more kinds of crosscutting features than JL's mixins. On the other hand, determining the value and the proper use of this additional flexibility is the subject of continuing research [64]. In addition, AspectJ must define precedence

rules to determine the order in which weaved code is executed, while the order of mixin application is implicit and clear in instantiations.

3.1.2 MULTI-DIMENSIONAL SEPARATION OF CONCERNS

Multi-Dimensional Separation of Concerns (MDSOC) [118] generalizes AOP (§3.1.1) in two ways. First, the MDSOC model addresses the evolution of all software artifacts, including requirements, documentation, and design, as well as code. Second, the MDSOC model does not create a new language to implement crosscutting features, but instead uses new techniques for composing code that is written in existing languages.

In MDSOC, different software artifacts are expressed in their own formalisms. For example, design is often expressed in UML [85], while code is expressed in a programming language. Each formalism separates concerns in its domain using its own composition and decomposition mechanisms. Concerns are implemented in the formalism's modules, which are composed to build complete artifacts.

The key insight of MDSOC is that all formalisms rely on a dominant decomposition method to create modules, and this reliance ultimately leads to the problems of crosscutting, tangling, and scattering that we described in the Motivation chapter. For example, class-based languages use data abstraction as their dominant decomposition method and classes as their modules. After a class hierarchy is established, features that crosscut existing classes cannot be easily encapsulated in a single module. For this reason, the *dominant dimension of concern* in object-oriented languages is data organization.

MDSOC defines *hyperslices* as sets of partially implemented modules. Hyperslices can encapsulate concerns in any dimension, especially non-dominant dimensions. Thus, in object-oriented code, hyperslices define crosscutting application features as sets of partially implemented classes. A *hypermodule* is a set of hyperslices and a *composition rule* that defines how the hyperslices are composed. Composition rules integrate hyperslices, which can be designed and implemented in isolation of each other. The main challenge in implementing MDSOC is defining composition rules that are both powerful and easy to use. The composition rules used in subject-oriented programming [51,88], which preceded MDSOC, provide a basis for the continuing research in this area.

Hyper/J [53] provides Java support for multi-dimensional separation of concerns. Hyper/J focuses on the adaptation, integration and on-demand remodularization of Java code. Hyperslices in Hyper/J can be mixed and matched to create customized applications. Hyper/J can also extract and, possibly, reuse feature code not originally separated into hyperslices. That is, Hyper/J supports the unplanned re-factorization of code to untangle feature implementations.

Java Layers can be viewed as an instantiation of the MDSOC model, just as it can be viewed as an aspect language. In JL, mixins and mixin layers are hyperslices, mixin compositions are hypermodules, and inheritance is the only composition rule. JL addresses only code artifacts and JL does not support unplanned code re-factorization as Hyper/J does. In JL, some design planning takes place in advance to ensure that components are composable with each other. On the other

hand, JL components are combined using inheritance, which is well-understood and does not require the development of new code composition techniques.

3.1.3 GENVOCA GENERATORS

The GenVoca model [15], which we described in §2.3.1, provides the conceptual foundation for Java Layers. GenVoca research has focused on developing domain-specific software generators that support the GenVoca model. The domains investigated in this research include database management systems [15], communication protocols [15], data structures [11,13,16,58], avionics [12], military command and control simulators [39], and compilers [14]. The number and breadth of these experiments reinforce the claim that the GenVoca model provides an effective foundation for software development.

There have also been two domain-independent GenVoca projects, both of which influenced JL research. The first project defined the P++ [100] language, which extends C++ to directly implement the GenVoca model. P++ uses the *realm* construct to define interfaces and the *component* construct to define reusable layers. An early version of Java Layers (§6.1) also directly implemented the GenVoca model, though our focus was on new language features that support mixin programming.

The second domain-independent GenVoca effort developed the idea of mixin layers [101,103] and then used them to build the Jakarta Tool Suite (JTS) [14]. JTS is a GenVoca generator that generates domain-specific languages. Mixins and mixin layers provide the technology foundation for software reuse in JL (§2.3).

JL departs from prior GenVoca research by focusing on the integration of mixin support in current object-oriented languages. Many ideas in JL, however, have precursors in previous GenVoca implementations. For example, to express the most derived type in a mixin-generated hierarchy, JTS uses a well-known name or a less general form of JL's **This** type parameter. Thus, the need for a most derived type was recognized before JL, but JL generalized its design and incorporated it into a domain-independent language.

3.2 Implementing Mixins

Most mixin research falls into one of two categories. The first category uses parametric polymorphism [29] to implement mixins. Under this approach, mixins and other parameterized types are typically treated as *type functions* or *type schemas*, which generate types, but are not themselves types. In languages that already support parametric types, adding mixins can be an almost trivial extension. Mixin research in this category often uses C++ [110] template classes, which already support mixins. Such research [101,38,126] emphasizes software engineering concerns and often includes experiments that test the effectiveness of different mixin programming techniques. Java Layers builds directly on this line of research, which we describe in detail in later sections.

The second category of mixin research defines *mixins as types*. Under this approach, mixin types extend their supertypes without relying on parametric polymorphism. The research [4,5,21,40] in this area focuses on the formal semantics of mixins and on the integration of mixins into existing type systems.

This integration typically uses the keyword *mixin* to declare new types, which either replace or work in conjunction with existing types (e.g., classes).

Recent mixin type proposals include two new programming languages. The first of these languages, JAM [3], has been implemented. JAM integrates mixin types into Java by adding two new keywords and by extending Java's type system. The JAM code below illustrates how mixin type `M` is declared and how it is used to define two subclasses (`Child1` and `Child2`) of two parent classes (`Parent1` and `Parent2`).

```
mixin M { member-declarations }
class Child1 = M extends Parent1;
class Child2 = M extends Parent2;
```

Since mixins are types in JAM, `Child1` is a subtype of both `M` and `Parent1` in the above code. `Child2` is also a subtype of `M`, which allows objects of types `Child1` and `Child2` to be treated as type `M` objects. In JAM, mixin inheritance can be constrained by specifying in mixin definitions the set of methods that superclasses must implement.

JAM puts a number of restrictions on how mixins are defined and used. For example, to preserve type system soundness, the keyword **this** cannot be used as a parameter in methods defined in mixins. Other restrictions in JAM include the inability to define constructors in mixins, the inability to express the most derived type in a mixin-generated hierarchy, and the inability to compose mixins with other mixins. This last restriction makes stepwise program refinement inconvenient in JAM.

The second recently proposed language extends Java with *mixin modules* [37], which are packaging constructs that contain class and mixin type declarations. This language supports extension and combination, which are composition operations that can be applied to modules, classes and mixins. Mixin modules are similar to JL mixins in two important ways. First, mixin modules capture the notion of a most derived type using the **This** keyword, which is analogous to JL's **This** type parameter (§4.3). Second, mixin modules enforce a kind of deep conformance (§4.2) by restricting mixin inheritance: Mixins can only extend mixins that have the same name and are inherited from another module. One limitation of mixin modules, however, is that the only constructor allowed in mixins is the implicit default constructor.

We conclude this section by mentioning two research proposals that do not fall into either of the above categories, but that are mixin-related. The first proposal, Jiazzi [76], implements a component system for Java that generalizes the concept of a Java package. Jiazzi components are parameterized modules that contain classes. In Jiazzi, mixin-style constructions are defined when the classes in a component inherit from the component's imported parameters. The second proposal defines *delegation layers* [89], which combine delegation [70] and virtual types [74] to change the behavior of a set of objects at runtime. In this context, delegation refers to dynamic, object-based inheritance. Delegation layers are similar to mixin layers, except mixin layers change the behavior of a set of classes statically.

CHAPTER 4 LANGUAGE FEATURES

In this chapter, we describe the Java Layers language features that address the programming challenges discussed in §2.4. We begin by discussing JL's implementation of constrained parametric polymorphism, which is provided for background purposes and does not represent a new contribution. We then discuss four novel JL language features, three of which have been implemented in our compiler.

Our design philosophy is to add simple, orthogonal features to Java to support mixin programming. Our goal is to limit the impact of these new features on Java and to limit unintended interactions between these features. Towards this goal, we add generics to Java, but we do not change Java's type system. We also add to Java two new modifiers, the **This** keyword, and two optional clauses in class declarations. Theses additions to Java are largely independent of each other, though the implicit **This** type parameter is closely integrated into instantiation mechanism for parametric types.

For each JL language feature, we present its design, algorithms, and implementation status, but we defer detailed discussion on implementation topics until Chapter 6. For now, we note that the JL compiler is a source-to-source compiler, so all transformations described in this chapter are implemented as source code transformations.

In citations, we use JLS to refer to the Java Language Specification [49] and JVMS to refer to the Java Virtual Machine Specification [71].

4.1 Constrained Parametric Polymorphism

This section describes JL's implementation of *constrained parametric polymorphism*, which is an enabling technology used by JL, but is not a new contribution. In §2.3.2, we described how parametric polymorphism, or generic types, provides a basis for the implementation of mixins. In §2.4.3, we described how code comprehensibility, robustness, and usability are enhanced when type parameter bindings can be constrained. We begin by discussing JL's parametric types and type parameter constraints. We then describe how type parameter scoping supports mixins and F-bounded polymorphism [28]. We conclude with related work in generic programming.

In JL, *parametric types* are class and interface declarations that are parameterized with *type parameters* or *primitive literal parameters*. Parametric types are *instantiated* when all type parameters and all primitive literal parameters are bound. Type parameters are bound to primitive or non-primitive types; primitive literal parameters are bound to literal values. The eight primitive literal types that can be used in parametric type declarations are boolean, byte, char, double, float, int, long, short. Table 2 shows three parametric types and example instantiations of each.

Parametric Types	Instantiations
class C<T; U; V>{...}	C<String, HashMap, int>, C<long, Vector, FileReader>
class D<T; int i; boolean b>{…}	D<String, 4, true>, D<double, 20, false>
interface I<T>{…}	I<StringBuffer>, I<boolean>

Table 2 - JL Generics and Instantiations

All formal parameters in parametric type declarations are uniquely named and are bound to actual parameters when the types are instantiated. Instantiated parametric types represent Java types and can be used wherever types are allowed in Java. Parametric types themselves, however, are not Java types and cannot be used in expressions unless they are properly instantiated. For example, class `E` below uses type parameters `T` and `U` on lines 3 and 4. The instantiation of `F` on line 3 is a valid type specification because `T` is bound whenever `E` is instantiated. The use of `G` on line 4, however, is invalid because `U` is not bound and the expression, `G<U>`, is not a Java type.

```
1. class E<T>
2. {
3.  F<T> f;  // OK, T is bound within E.
4.  G<U> g;  // Error, U is not bound.
5. }
```

In JL, anonymous classes cannot be parameterized due to their spare syntax. In addition, JL does not support parameterized local classes.

4.1.2 TYPE PARAMETER CONSTRAINTS

JL uses the two ways to inherit in Java to define two kinds of constraints on type parameters. JL uses an *extends constraint clause* to guarantee that a type parameter binding extends one or more types. JL uses an *implements constraint clause* to guarantee that a type parameter binding implements one or more interfaces. These clauses restrict the bindings of type parameters to subtypes of the types specified in the constraints.[1] We now illustrate the use of constraint clauses by presenting example type declarations.

In class `C` below, the extends clause requires that the type bound to `T` be a subtype of `FileReader`. Similarly, `C`'s implements clause requires that the class bound to `U` implements the `Serializable` interface. In interface `I`, the type bound to `V` must be a subclass of `LinkedList` that also implements the `Runnable` interface.

```
class C<T extends FileReader; U implements Serializable>{…}
interface I<T extends LinkedList implements Runnable>{…}
```

In JL, multiple constraints can be specified in each clause. Class `D` below illustrates this more complex case. The class bound to `T` must subclass both `LinkedList` and `AbstractSequentialList`. This bounded class must also implement the `Serializable` and `Runnable` interfaces.

[1] For constraint checking purposes, a type is considered a subtype of itself.

```
class D<T extends LinkedList, AbstractSequentialList
          implements Serializable, Runnable> {…}
```

4.1.3 TYPE PARAMETER SCOPE

In a JL type declaration, a type parameter's scope includes the type parameter declaration clause (i.e., everything between the angle brackets), any inheritance clauses, and the body of the type. In addition, type parameters cannot be hidden in the body of their declaring type by nested types or by other type parameters. Thus, if type *C* declares type parameter *T*, then no type or type parameter can be declared in the body of *C* with the name *T*.

Mixins are supported by using type parameters in inheritance clauses. Below, class `J` and interface `K` represent simple mixins. `L` and `M` represent more complex declarations that are also possible in JL.

```
class J<T> extends T {…}
interface K<T> extends T {…}
class L<T; U> extends T implements U {…}
class M<T> extends A<T> {…}
```

JL also supports *F-bounded polymorphism* [28], which allows the specification of recursively constrained type parameters. Below in class `N`, type parameter `T` is constrained by interface `F`, which is itself dependent on, or a function of, `T`. Class `O` illustrates how type parameters can be used in a declaration clause before they are defined.

```
class N<T implements F<T>> {…}
class O<T implements G<U>; U implements H<T>> {…}
```

4.1.4 RELATED WORK

In the previous section, we summarized JL's implementation of constrained parametric polymorphism. In this section, we describe other implementations of parametric polymorphism and discuss proposals for adding parametric polymorphism to Java. We also compare JL to one of the more prominent of these proposals.

Support for parameterized programming [48] first appeared in programming languages in the late 1970's. This support includes the use of parameterized modules in OBJ [43,47], Ada [6] and CLU [72], the last of which also allows constraints on type parameters. At about the same time, ML [78,124] also provided support for parameterized types.

More recently, the widespread use of C++ [110] has helped popularize parameterized programming. C++ templates and important libraries that use templates, such as the Standard Template Library [107,110], have introduced parameterized types to large numbers of programmers. The C++ template specification defines a Turing-complete language [36] and supports advanced features like mixins, partial evaluation, manual template specialization, and lazy code generation. Czarnecki and Eisenecker [36] use templates for meta-programming, which allows them to generate highly configurable applications. The drawbacks of using C++ templates include the lack of type parameter constraints, code bloat when too

many instantiations are specified, and the difficulty of tracing compilation errors in template code.

C++ is significant in mixin research because it provides a well-supported platform for experimentation and a large community of programmers. VanHilst and Notkin [125,126,127] used C++ mixins to explore new design techniques and new ways to increase code reuse. Smaragdakis and Batory [101,103] used C++ to demonstrate that mixin layers (§2.3.3) and stepwise program refinement (§2.3.1) further increase the benefits of mixin programming. They also developed a number of techniques specifically for programming with mixins in C++ [104].

As a result of this research, we recognized that C++ is missing important support for mixin programming. C++ provides mixins, but it does not provide mechanisms that make programming with mixins convenient. JL's new language features address these deficiencies. There is, however, one area in which C++ support for mixin programming is well-developed: Existing C++ compilers can already produce efficient code from mixins. Berger, Zorn and McKinley [17] show that even though mixins introduce many layers of abstraction, aggressive method inlining results in mixin-generated code that performs at least as well as, and sometimes better than, conventional code. This result reinforces our optimism about the potential effectiveness of the class hierarchy optimization described in Chapter 5.

Much of the recent activity in parametric polymorphism focuses on Java. A number of proposals have been made for implementing parametric polymorphism in Java [1,19,24,33,82,105], though only one of these proposals [1] sup-

ports mixins. Most of these proposals implement type parameter constraints similar to the way JL does. The PolyJ proposal [82], however, takes a structural approach by specifying type parameter constraints as lists of required methods. The Generic Java (GJ) [24] proposal forms the basis of JSR-14 [56], which is the Java Specification Request to add generic types to the Java language. Given the importance of the GJ approach to the Java community, we briefly compare it to JL.

4.1.4.1 Java Layers and Generic Java

In this section, we explore the fundamentally different approaches to parametric polymorphism that JL and GJ each implement. We first describe GJ's *homogeneous* approach, its implementation, and its limitations. We then describe JL's *heterogeneous* approach and compare it to GJ.

GJ is a backward compatible extension of Java that implements parameterized types and methods. Backward compatibility means that existing libraries can be retrofitted with generic interfaces without changing the library code. GJ uses a *homogeneous* implementation, which means that all instantiations of a parametric type execute the same compiled code. This implementation eases the transition from legacy code because parameterized and unparameterized versions of the same types can co-exist in an application without code duplication. In general, homogeneous implementations are memory efficient because a single class implements all instantiations of a parametric type.

Homogeneous implementations, however, also have a number of disadvantages. To understand these disadvantages, we describe GJ's implementation. GJ works by *erasing* type parameters at compile time and replacing them with

general types that are appropriate for all instantiations. Figure 6 shows parametric class `C` and its erasure, which gets compiled. In GJ, no type parameter information is available at runtime. Instead, the GJ compiler inserts dynamic type casts into code to guarantee type safety; it also inserts bridge methods to guarantee that method overriding works properly.

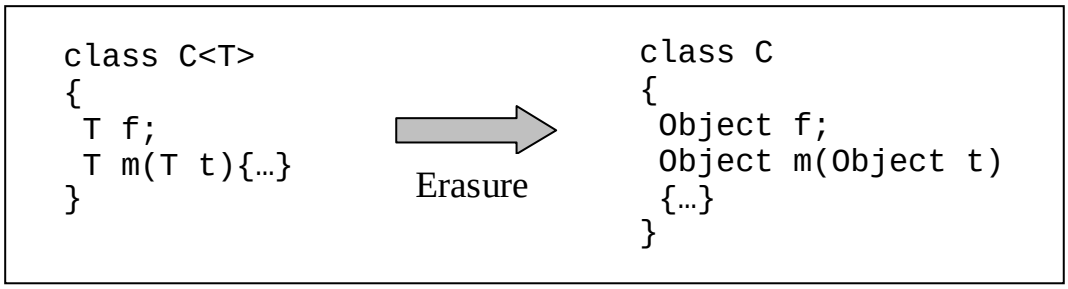


Figure 6 - Type Parameter Erasure in Homogeneous Implementations

Type erasure loses information because actual type parameter bindings known at compile time are not available at runtime. In general, erasure restricts type parameter usage whenever a specific type is needed at runtime. For example, a type parameter cannot be used as the non-array type allocated in a **new** expression. Once an actual type parameter is replaced with its more general erasure type, the actual type is not known at runtime and cannot be allocated. For the same reason, type parameters cannot be used as the type in cast, **catch** or **instanceof** expressions. More subtly, type erasure restricts a class from inheriting (directly or indirectly) from two different instantiations of the same parametric interface. In GJ, type erasure also prevents the binding of primitive types to type parameters.

Homogeneous implementations impose a number of other usage limitations on the programmer, three of which we list here.

First, the type parameters of a parametric class cannot be used inside the class's static initializers, static field declarations, or static methods. This limitation is necessary because all instantiations of a parametric type share the same compiled code. This code sharing implies that each static definition must work in all instantiations, which means that these definitions cannot depend on instantiation-specific type parameters.[2] GJ softens the impact of this limitation by supporting parameterized static methods, which are static methods that declare their own type parameters.

Second, in homogeneous implementations like GJ that do not use type parameter information at runtime, reflection can be used to circumvent the dynamic type checks inserted by the compiler. Consider, for example, parametric class `C` and its erasure in Figure 6 above. If we write a program that uses `C<String>`, the compiler will insert dynamic type casts into our program. These type casts guarantee that accesses to field `f` and calls to method `m` use `String` values as required. In the same program, however, we could use reflection to directly access `f` or directly execute `m` using any non-primitive type that we choose.

Third, and the most significant limitation from the perspective of Java Layers, is that *homogeneous implementations in Java cannot support mixins.* When GJ performs type parameter erasure, it loses the supertype information nec-

[2] GJ also prohibits the use of a parametric class's type parameters in the class's static member types. *Static* means *single definition* when applied to fields and methods, but it means *no lexically enclosing instances* (JLS §8.5.2) when applied to nested types. Static nested types do not share code between instantiations any differently than top-level types or inner classes, so there's no need for GJ's restrictions on the use of type parameters in static nested types.

essary for mixin instantiation. Moreover, it is not possible to use the same compiled class for all instantiations of a mixin because different instantiations can have different supertypes, and supertypes are fixed in Java bytecode [71].

This brings us to the alternative implementation of parametric types that is used in JL. JL uses a *heterogeneous* implementation, which means that each distinct instantiation of a parametric type executes its own specialized version of compiled code. Figure 7 shows parametric class `C` and the instantiation of `C<String>`, which gets compiled. Since each instantiation generates specialized code, heterogeneous implementations like JL and C++ can experience code bloat if a large number of instantiations are used.

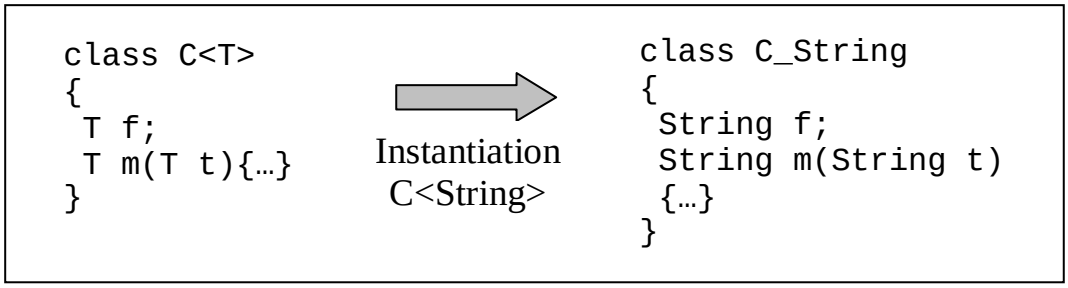


Figure 7 - Instantiation in Heterogeneous Implementations

There are a number of interesting points of comparison between JL and GJ. First, none of the restrictions on type parameter usage in GJ applies in JL, which makes JL somewhat more expressive than GJ and, possibly, simpler to program because there are fewer rules to remember. Second, primitive types can be bound to type parameters in JL. Third, dynamic type casts are not inserted into code by the JL compiler to augment type checking. As a result, reflection cannot

be used in JL to circumvent type checking because type parameter information is incorporated into each generated class.

On the other hand, JL does not support the backward compatibility properties of GJ, so the migration to parameterized Java could be more difficult using JL. In addition, JL requires name mangling (§6.3) to uniquely identify each generated class, which exposes users to compiler generated names.

Lastly, like all heterogeneous implementations in Java, JL is susceptible to the *package problem* [86]. To understand this problem, consider the instantiation *R.C<S.D>*, where *R* and *S* are different packages, *C* is a parameterized class, and *D* is a class. The class generated by this instantiation, *C_D*, must reside in some package. We can put *C_D* in *R* only if *D* is public. We can put *C_D* in *S* only if *C* does not refer to package members in *R*. If neither of these conditions holds, we cannot put *C_D* in *R*, *S*, or any other package, and the instantiation fails.

In JL, all generated classes are put in the parametric type's package. Consequently, actual type parameters are either public or they reside in the same package as the parametric type in which they are used. An interesting future research topic would explore how heterogeneous implementations of parametric polymorphism could be smoothly integrated into a language's access control mechanism. The goal would be to eliminate inconveniences like the package problem.

4.2 Deep Conformance

JL's support for deep conformance promotes the use of mixin layers and stepwise program refinement as described in §2.4.3. Deep conformance is the general term we use to groups together the concepts of *deep subtyping* and *deep interface conformance* [101]. In this section, we define both of these concepts and describe JL's novel support for them. We then discuss work related to deep conformance.

4.2.1 DEEP SUBTYPING

There are two kinds of deep subtyping in JL. The first kind applies to classes and was originally defined by Smaragdakis [101]. The second kind applies to Java interfaces. We recursively define both kinds below.

> Class *C* is a *deep subclass* of another class *B* if (1) *C* is a subclass of *B*, and (2) for every publicly accessible nested class *B.N*, there is a publicly accessible nested class *C.N* that is a *deep subclass* of *B.N*.
>
> Interface *J* is a *deep subinterface* of another interface *I* if (1) *J* is a subinterface of *I*, and (2) for every nested interface *I.N*, there is a nested interface *J.N* that is a *deep subinterface* of *I.N*.

Since interface members are always public, the above two definitions are the same in all respects except that one describes classes and the other describes interfaces. In this discussion, statements we make about classes to also apply to interfaces and vice versa.

Deep subclassing characterizes the relationship between a superclass *B* and its deep subclass *C* in three ways. First, for each public member class in *B*,

there is public member class with the same name in *C*. These member classes in *C* can be either inherited or explicitly declared in *C*. Second, if a public member class is declared in *C* and that class has the same name as a public member class in *B*, then the class in *C* is a subclass of the class in *B*. Third, deep subclassing is applied recursively to all levels of nesting in *B* and *C*.

Figure 8 shows class `D` and its deep subclass `E`. The nested classes on the `D`'s public interface are `X`, `X.X1`, `Y` and `Z`. All like-named classes on `E`'s public interface are subclasses of their corresponding classes in `D`. If `E.X`, `E.X.X1`, or `E.Y` did not inherit from their corresponding class in `D`, or if `E.Z` was defined and it did not inherit from `D.Z`, then `E` would not be a deep subclass of `D`.

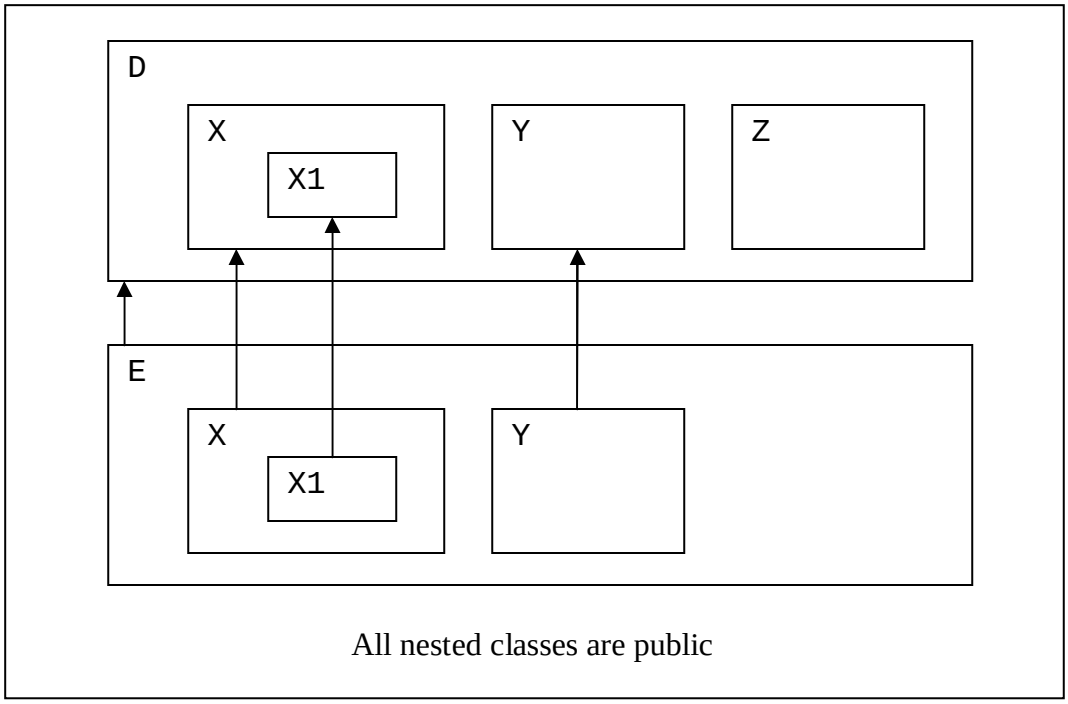


Figure 8 - Deep Subclassing

JL provides the **deeply** modifier to enforce deep subtyping constraints. Programmers specify deep subtypes by using **deeply** in *extends clauses*. Figure 9 shows how **deeply** is used to restrict inheritance in mixin layers. Classes `A` and `B` contain nested classes. Mixin class `F` is defined to be a deep subtype of the class that gets bound to type parameter `T`. In instantiation `F<A>`, `F<A>.Inner1` extends `A.Inner1` as required, so instantiation succeeds. In instantiation `F<B>`, however, `F<B>.Inner2` does not extend `B.Inner2` type, so instantiation fails because of a deep conformance error.

```
class A {public class Inner1 {…}}
class B {public class Inner1 {…} public class Inner2 {…}}

class F<T> extends T deeply {
 public class Inner1 extends T.Inner1 {…}
 public class Inner2 {…}
}

Instantiations
F<A> // OK, F<A> is a deep subclass of A
F<B> // Compile error, F<B> is not a deep subclass of B
```

Figure 9 - Restricting Inheritance with Deep Subtyping

Figure 10 shows how the **deeply** modifier is used to constrain type parameter bindings. In class `G`, `T` can only be bound to classes that are deep subtypes of `A` (including `A`). In mixin `H`, **deeply** is used in both the constraint clause and the inheritance clause. In the constraint clause, `U`'s binding is restricted to classes that are deep subtypes of `A`. In the inheritance clause, all instantiations of `H` are deep subclasses of the class bound to `U`. Together, these two uses of **deeply** imply that all instantiations of `H` are deep subtypes of `A`.

```
class A {public class Inner1 {…}}
class G<T extends A deeply> {…}
class H<U extends A deeply> extends U deeply {…}
```

Figure 10 - Constraining Type Parameters with Deep Subtyping

The position of the **deeply** modifier in an extends clause affects its meaning. When **deeply** appears after a type or type parameter, it applies only to its immediate antecedent. When **deeply** appears directly after the *extends* keyword, it applies to all types and type parameters in the clause. Using these rules, the two interface definitions in Figure 11 are equivalent.

```
interface I<T; U> extends T deeply, U deeply {…}
interface I<T; U> extends deeply T, U {…}
```

Figure 11 - Deep Subtyping Syntax

4.2.2 DEEP INTERFACE CONFORMANCE

Deep interface conformance is similar to deep subtyping, except that deep interface conformance relates interfaces to the classes that subtype them. In this section, we define deep interface conformance and then show how it is used to specify nested interfaces as prototypes for nested classes.

We have adapted the definition of deep interface conformance from Smaragdakis's original definition [101]. Our recursive definition below uses Java terminology.

> Class *C conforms deeply* to interface *I* if (1) *C* implements *I*, and (2) for each publicly accessible nested interface *I.N*, there is a publicly accessible class *C.N* that *conforms deeply* to *I.N*.

Deep interface conformance characterizes the relationship between an interface *I* and its deeply conforming class *C* in two ways. First, for each member interface in *I*, there is public member class with the same name in *C* and that class implements the member interface. Second, deep interface conformance is applied recursively to all levels of nesting in *I* and *C*.

Figure 12 shows interface `I` and its deeply conforming class `C`. The nested interfaces on `I`'s public interface are `X1`, `X1.X1a`, and `Y2`. All like-named classes on `C`'s public interface implement their corresponding interface in `I`. If `C.X1`, `C.X1.X1a`, or `C.Y2` did not exist, or if they did not implement their corresponding interface in `I`, then `C` would not deeply conform to `I`.

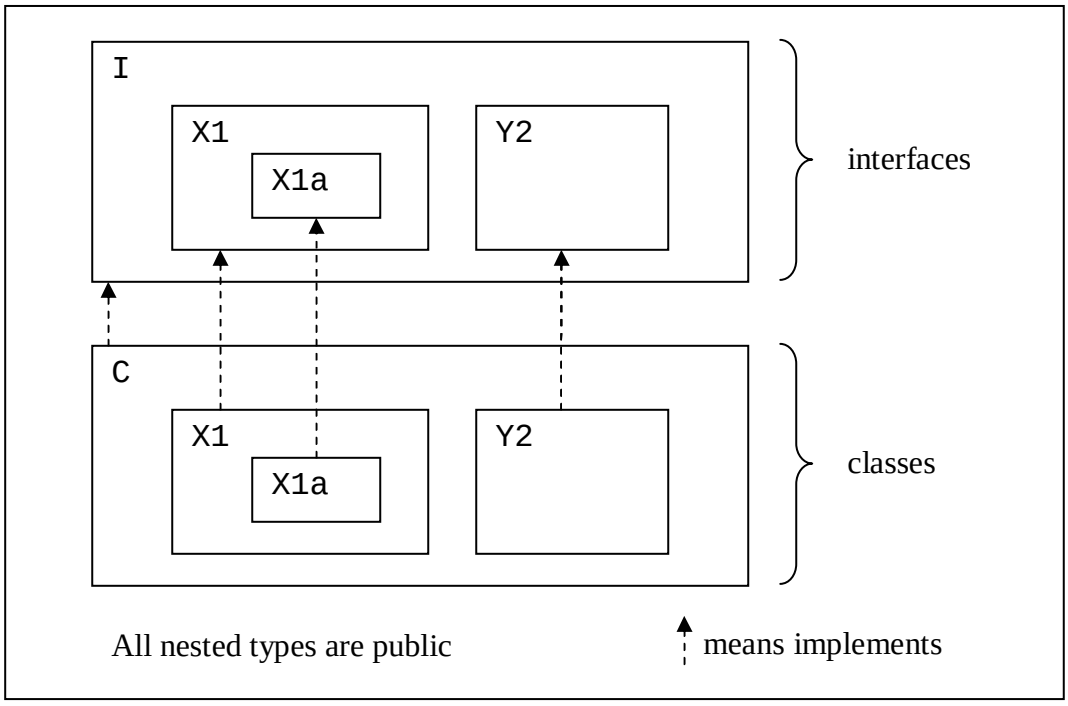


Figure 12 - Deep Interface Conformance

JL uses the **deeply** modifier introduced in the previous section to enforce deep interface conformance. Figure 13 shows how **deeply** is used in *implements clauses* to restrict inheritance in mixin layers. Interfaces `J` and `K` contain nested interfaces. Mixin class `F` is defined to deeply conform to the interface bound to type parameter `T`. In instantiation `F<J>`, `F<J>.Inner1` implements `J.Inner1` as required, so instantiation succeeds. In instantiation `F<K>`, however, `F<K>.Inner2` does not implement `K.Inner2` type, so instantiation fails because of a deep conformance error.

```
interface J {interface Inner1 {…}}
interface K {interface Inner1 {…} interface Inner2 {…}}

class F<T> implements T deeply {
 public class Inner1 implements T.Inner1 {…}
 public class Inner2 {…}
}

Instantiations
F<J> // OK, F<J> deeply conforms to J
F<K> // Compile error, F<K> does not deeply conform to K
```

Figure 13 - Restricting Inheritance with Deep Interface Conformance

Figure 14 shows how the **deeply** modifier is used to constrain type parameter bindings. In class `G`, `T` can only be bound to classes that deeply conform to interface `L`. In mixin `H`, **deeply** is used in both the constraint and inheritance clauses. In the constraint clause, `U`'s binding is restricted to classes that deeply conform to `L`. In the inheritance clause, all instantiations of `H` are deep subclasses of the class bound to `U`. Together, these two uses of **deeply** imply that all instantiations of `H` also deeply conform to `L`. The placement of **deeply** in implements clauses follows the same syntactic rules as described in §4.2.1 for extends clauses.

```
interface L {interface Inner1 {…}}

class G<T implements L deeply> {…}

class H<U implements L deeply> extends U deeply {…}
```

Figure 14 - Constraining Type Parameters with Deep Interface Conformance

4.2.3 EXPLICITLY PROPAGATING TYPES

In previous sections, we defined deep subtyping and deep interface conformance in terms of public nested types. We also defined them as independent, non-overlapping constraints on the nested structure of types. In this section, we broaden the capabilities of deep conformance in two ways. First, we allow package or protected types to participate in deep conformance processing. Second, we allow the processing of deep subtyping constraints and deep interface conformance constraints to be interleaved with each other.

We introduce a new modifier to extend the capabilities of deep conformance. The **propagate** modifier allows nested types to be explicitly included in deep conformance processing.[3] Deep conformance, whether it is deep subtyping or deep interface conformance, processes explicitly propagated nested types in addition to the public nested types that are always processed. We now describe the two ways that this extension is used.

The first reason to use **propagate** is to include package or protected types in deep conformance processing. Figure 15 shows class `A`, which contains protected class `InnerC`. Mixin `F` can be instantiated with `A` because (1) `F<A>.InnerC` is a subclass of `A.InnerC`, and (2) `F<A>.InnerC` has access control at least as permissive as the access control on `A.InnerC`. On the other hand, `G<A>` fails to compile because `G<A>.InnerC` is not a subclass of `A.InnerC`.

[3] JL's **propagate** modifier is used for both constructor propagation (§4.4) and deep conformance.

```
class A {propagate protected class InnerC {…} }

class F<T> extends T deeply {
 protected class InnerC extends T.InnerC {…}
}

class G<T> extends T deeply {
 protected class InnerC {…}
}

Instantiations
F<A> // OK, F<A> is a deep subclass of A
G<A> // Compile error, G<A> is not a deep subclass of A
```

Figure 15 - Propagating Non-Public Nested Types

The second reason to use **propagate** is to interleave deep subtyping and deep interface conformance. Figure 16 shows class `B`, which contains package accessible interface `InnerI`. Mixin `M` can be instantiated with `B` because (1) `M<B>.InnerI` implements `B.InnerI`, and (2) `M<B>.InnerI` has access control at least as permissive as the access control on `B.InnerI`. On the other hand, `N<B>` fails to compile because no class `N<B>.InnerI` is defined that implements `B.InnerI`. **propagate** can be used in a similar way to deeply subtype classes nested in interfaces.

```
class B {propagate interface InnerI {…} }

class M<T> extends T deeply {
 class InnerI implements T.InnerI {…}
}

class N<T> extends T deeply {}

Instantiations
M<B> // OK, M<B> is a deep subclass of B
N<B> // Compile error, N<B> is not a deep subclass of B
```

Figure 16 - Mixing Deep Subtyping and Deep Interface Conformance

4.2.4 DISCUSSION

Adding deep conformance to Java provides support for a currently under-specified part of the language. In Java, a class that implements an interface is not required to implement that interface's nested interfaces [49,112]. For example, Figure 17 shows four classes that implement interface `I`, which has a nested interface `Inner`. These classes all have a member type named `Inner`, but this member type is different for each class: `C.Inner` is the inherited interface `I.Inner`; `D.Inner` is a newly defined interface that hides `I.Inner`; `E.Inner` is a class that implements `I.Inner`; and `F.Inner` is a class that hides `I.Inner`.

```
interface I {interface Inner {…} }

class C implements I {}
class D implements I {interface Inner {…} }
class E implements I {class Inner implements I.Inner {…} }
class F implements I {class Inner {…} }
```

Figure 17 - Implementing Interfaces in Java

The above example shows that interface implementation in Java is *shallow*, which means that Java classes do not have to implement their interfaces' nested interfaces. Deep interface conformance provides a way for Java programmers to specify exactly how an interface's nested interfaces are to be treated. In Figure 17, class `E` deeply conforms to interface `I`; the use of **deeply** in the definition of `E` would both advertise and enforce this conformance property.

The most important benefit of deep conformance, however, is its enhanced support for mixin programming. Deep subtyping and deep interface conformance reinforce the use of mixins as incremental refinements by establishing a uniform pattern of inheritance in mixin compositions. This uniformity has two benefits. First, applications can be built in a stepwise manner because features are implemented in mixin layers that have a common structure. This common structure allows the mixin layers to be composed with one another. Second, mixin-generated code presents predictable interfaces to users because nested type names are preserved in subclasses.

Figure 18 illustrates how deep conformance promotes well-structured code in the Fidget family of graphical user interface libraries (§7.2). `BaseFidget` and

`LightWeightFidget` are simplified versions of Fidget classes that we introduced in Figure 5 in §2.4.3. The Fidget design rests on three pillars. The first pillar is the `FidgetTkIfc` interface, which declares all the nested widget interfaces. The second pillar is `BaseFidget`, which deeply conforms to `FidgetTkIfc` and provides the basic widget function used in all mixin compositions. `Checkbox` and `Button` are examples of two widgets in `BaseFidget` that implement their corresponding interfaces from `FidgetTkIfc`. The third pillar of Fidget design is the use of mixins layers that incrementally add function to libraries. `ColorFidget`, which adds color display support to a GUI library, represents the typical structure of a Fidget mixin layer. In Fidget, every mixin's supertype deeply conforms to `FidgetTkIfc` and every mixin deeply subtypes its supertype. The three inheritance clauses in `ColorFidget` that name `FidgetTkIfc` are included for clarity, but strictly speaking are not necessary.

```
interface FidgetTkIfc {
 interface Checkbox {…}
 interface Button {…} …
}

class BaseFidget<> implements FidgetTkIfc deeply
{
 public abstract static class Checkbox
  implements FidgetTkIfc.Checkbox {…}

 public abstract static class Button
  implements FidgetTkIfc.Button {…} …
}

class ColorFidget<T implements FidgetTkIfc deeply>
 extends T deeply
 implements FidgetTkIfc deeply
{
 public static class Checkbox
  extends T.Checkbox
  implements FidgetTkIfc.Checkbox {…}

 public static class Button
  extends T.Button
  implements FidgetTkIfc.Button {…} …
}
```

Figure 18 - The Use of Deep Conformance in Fidget

The three pillars of Fidget design provide a foundation on which new Fidget function can be built. New Fidget features are implemented in mixin layers that have the same structure as `ColorFidget`. These mixin layers are syntactically interchangeable since they all deeply conform to the same interface. In §7.2, we describe how different mixin compositions generate different GUI libraries. From a design perspective, Fidget libraries can be built using stepwise refinement because each widget class inherits from its corresponding widget class

in its supertype mixin layer. In addition, different Fidget libraries present the same basic interface to user applications (though new features can add new methods to widget classes).

4.2.4.2 Design and Implementation Alternatives

This section describes alternate ways to implement deep conformance. We first discuss how the timing of conformance checking affects the semantics of deep conformance. We then describe how different levels of automation affect how programmers use nested interfaces. In both cases, we describe the approach taken by JL.

The first implementation issue is to decide when deep conformance processing is performed. Deep conformance processing can always take place before runtime because it checks static class structure. Processing can occur either when types are defined (compile-time) or when type parameters are bound (instantiation-time). In some cases, the kind of constraint determines when conformance processing occurs. For example, constraints on type parameters can only be checked at instantiation-time because that is when type parameter bindings are known. Similarly, constraints on inheritance clauses of non-parametric types can only be checked at compile-time because these types are not processed at instantiation-time.

The interesting case, however, is deciding when constraints on inheritance clauses of parametric types should be checked. In this case, checks can be performed at either instantiation-time or at compile-time, and the time chosen determines the extent of the checking that occurs. Timing plays such a pivotal role be-

cause different type information is available at different times. If checking occurs at instantiation-time, then the actual types bound to type parameters can be used. If checking occurs at compile-time, then type parameter constraints are often the only type information available for conformance checking.

The timing of the conformance checks affects whether a parametric type instantiation succeeds or not. To understand this effect, consider mixin `M` and its constrained type parameter `T` shown in Figure 19. The figure also contains an instantiation of `M` using class `A`, which is not shown. Conformance checking that involves the second occurrence of **deeply**, the occurrence in `M`'s inheritance clause, is affected by timing. At instantiation-time, deep subclass processing can inspect all nested types in `A`, including nested types not in `ConstraintClass`. At compile-time, however, deep subclass processing can only inspect nested types in `ConstraintClass`, which are a subset of the nested types in any actual type bound to `T`. A similar example could be constructed for deep interface conformance.

```
class M<T extends ConstraintClass deeply>
  extends T deeply {…}

Instantiation
M<A>
```

Figure 19 - Deep Subclass Testing

The compile-time approach relies on type parameter constraints to perform deep conformance checking. These constraints are used when inheritance clauses contain type parameters as we saw above. If no constraints are specified,

however, then inheritance clauses like those in class `M` cannot be checked. If the constraints are themselves parameterized, then deep conformance processing becomes complicated.

For these reasons, JL implements the instantiation-time approach, which relies on the actual types that are bound to type parameters. This approach uses more precise type information than is available at compile-time. Unfortunately, this extra precision can sometimes cause deep conformance violations that cannot be foreseen when a parametric class is defined. For example, violations can occur when a mixin and its supertype use the same name for unrelated nested types. In practice, these name conflicts should be rare because applications use deep conformance as part of a coordinated design, as we saw in the Fidget example in the last section. The key point, however, is that instantiation-time conformance uses specific type information to perform more rigorous checks.

The second implementation question involves choosing the proper level of automation for deep interface conformance. In §4.2.2, we specified that if class *C* deeply conforms to interface *I*, then *C* must declare or inherit an appropriately named nested class that implements each nested interface in *I*. If one of these nested classes is not found during conformance checking, the compiler could either report an error or generate the missing nested class. The error reporting approach is used in the current version of JL; the generative approach was used in an earlier version of JL (§6.1). We now compare these two approaches.

The generative approach reduces programmer effort, similar to the way that automatically generated default constructors reduce effort. This approach

also reduces the number of conformance violations because classes are automatically generated when needed. On the other hand, the error reporting approach requires that programmers be familiar with the interfaces they use, which is a reasonable way to promote good programming practice. We prefer the error reporting approach because of its simplicity, but both approaches have merit.

In summary, we described some of the design tradeoffs that confront JL and other implementations of deep conformance. The main point is that even though deep conformance can be precisely defined as a set of relationships between types, its actual semantics depend on implementation details that also need to be precisely defined. We now describe other approaches to deep conformance.

4.2.5 RELATED WORK

The notion of deep subtyping was implicitly part of the *realm* definition of P++ [100], a 1996 GenVoca extension to C++ that used nested classes to achieve stepwise program refinement. In 1998, deep subtyping was explicitly defined in the private correspondences of Wadler, Odersky and Smaragdakis concerning Generic Java [129]. Subsequently, deep interface conformance was defined by Smaragdakis [101,102] as a way to express properties about classes that implemented nested interfaces.

JL adopts the formal definitions of deep subtyping and deep interface conformance used by Smaragdakis [101]. JL contributes by providing an alternative to the design proposed by Smaragdakis and by providing the first implementation of deep subtyping and deep interface conformance. In the last section, we saw how implementing deep conformance exposes certain design issues and we saw

how JL addresses these issues. These same issues apply to Smaragdakis's design, though the design specification does not explore them. We now compare JL and Smaragdakis's design proposal.

The goal of Smaragdakis's design is to provide the essentials of deep interface conformance in Java, to support Java's current interface semantics, and to avoid adding new keywords to Java. This design can be easily adapted to extensions of Java that support constrained parametric polymorphism, like Generic Java [24].

Figure 20 shows interface `I` that contains nested interface `InnerI`. Smaragdakis introduces the concept of *class prototypes*, which are analogous to method prototypes and can appear in interface definitions. The class prototype in `I` requires that any class that implements `I` must also contain a nested class named `NestedC`, and `NestedC` must implement `InnerI`. Class `C` in the example meets this requirement.

```
interface I {
 interface InnerI {…}

 class NestedC implements InnerI; // class prototype
}

class C implements I {
 public class NestedC implements I.InnerI {…}
}
```

Figure 20 - Deep Interface Conformance using Class Prototypes

The use of class prototypes is simple and similar to the use of method prototypes, which Java already supports. The class prototype in `I` above specifies the

name of the required nested class in `C` (`NestedC`), but this name does not have to be same as `I`'s nested interface (`InnerI`) as it does in JL. Also, class prototypes do not require new keywords; JL requires the new **deeply** modifier.

The most significant difference between the two approaches, however, is how the decision to apply deep conformance is made. When using class prototypes, the interface writer decides whether or not all classes that implement an interface will deeply conform to it. On the other hand, the class writer in JL decides whether a class deeply conforms to an interface or whether it implements an interface using standard Java semantics. In this sense, the JL approach is more flexible because classes can deeply conform to any interface, whether or not the interface writer originally planned for deep conformance.

JL's approach to deep interface conformance automatically implements the recursive aspect of the definition (§4.2.2). Deep conformance in JL is automatically recursive to all nesting levels whenever **deeply** is used. On the other hand, deep conformance using class prototypes must be explicitly specified at each nesting level in the interface source code.

Lastly, we note that JL uses the **propagate** keyword to process non-public nested types and to mix the processing of deep subtyping and deep interface conformance. Similar capabilities could be added to the class prototype approach, but they are not currently specified.

4.3 The Implicit *This* Type Parameter

JL's implicit **This** type parameter allows programmers to express the most derived class of a mixin-generated hierarchy. In §2.4.2, we described how **This** extends the semantics of Java's self-type references to mixin types. In this section, we define JL's implicit **This** type parameter and describe its novel design and implementation. The idea of implicit type parameters, and **This** in particular, can be adapted to other parametrically polymorphic object-oriented languages.

We begin by describing what **This** means and how it is used in self-type references. We then show how **This** addresses a more general problem in object-oriented languages. After these motivating examples, we precisely define **This** by specifying how it is bound. We then discuss design and implementation topics, and we conclude with related work.

4.3.1 THE SEMANTICS OF *THIS*

JL's **This** type parameter is conceived in the spirit of Java's *this* reference. In Java, the *this* object reference is used in non-static methods to refer to the object on which the method is invoked. The *this* reference is an implicit argument passed to all constructors and non-static methods. In JL, the **This** type parameter is used in parametric types to refer to the most derived class instantiated by a composition of parametric types. The **This** type parameter is an implicit type parameter passed to all parametric types.

In each instantiated parametric type, **This** is bound to a class type. Depending on the context, we use the term *This-binding* to refer either (1) to the process of binding **This** to a class type or (2) to the class type that actually gets

bound to **This**. In §4.3.3, we specify the rules for binding **This**. These rules define special This-bindings for parametric interfaces and for mixin types. In the remainder of this section, we use two examples to introduce the capabilities of **This**.

The top portion of Figure 21 shows the definition of class `C`, which explicitly declares type parameter `T`. The type parameters `T` and **This** are both used in the definition of `C`. The bottom portion of Figure 21 shows an instantiation of `C` with `String`. The instantiated class is named `C_String`, which follows a naming scheme that appends the actual type parameter names in declaration order to the parametric type name.[4] `C_String` is the most derived class generated by this instantiation of `C`, so `C_String` is bound to **This** in the generated code

[4] The generated names shown in this section are simplified JL names; see §6.3 for details.

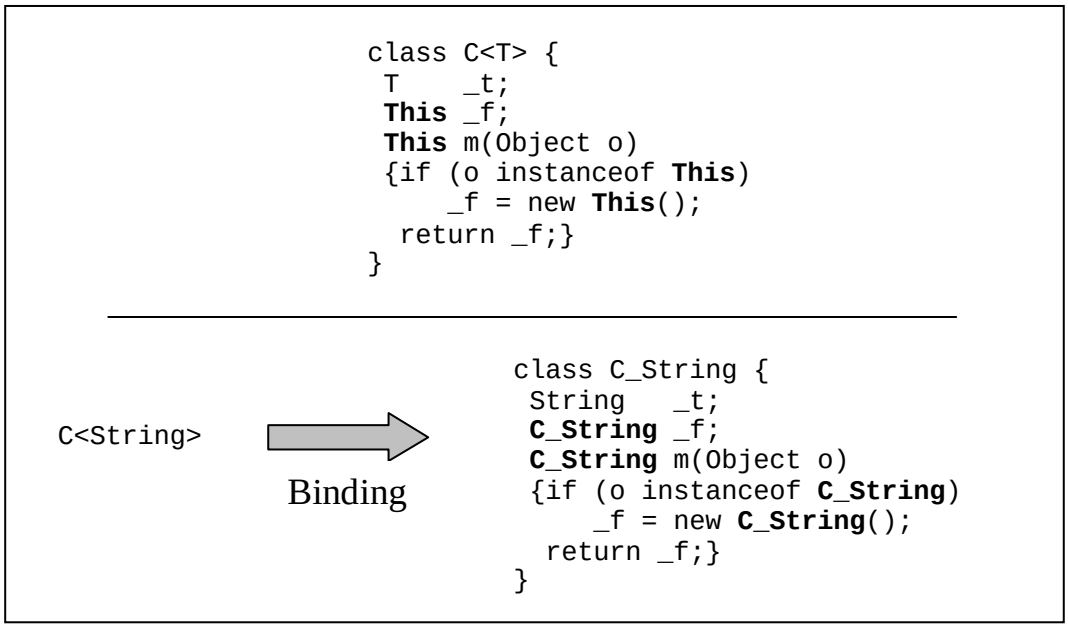


Figure 21 - Binding **This** in a Parametric Class

The second example describes This-binding in mixin classes. The top portion of Figure 22 defines two classes. The parametric class `D` declares no explicit type parameters but uses **This** in its definition. In JL, parametric types can access their implicit type parameter even if they do not explicitly declare type parameters. Mixin class `M` is defined and it also uses **This**. The bottom portion of Figure 21 shows an instantiation using `M` and `D`. The This-binding in both `M` and `D` is `M_D_`, which is the most derived class in the mixin-generated hierarchy.

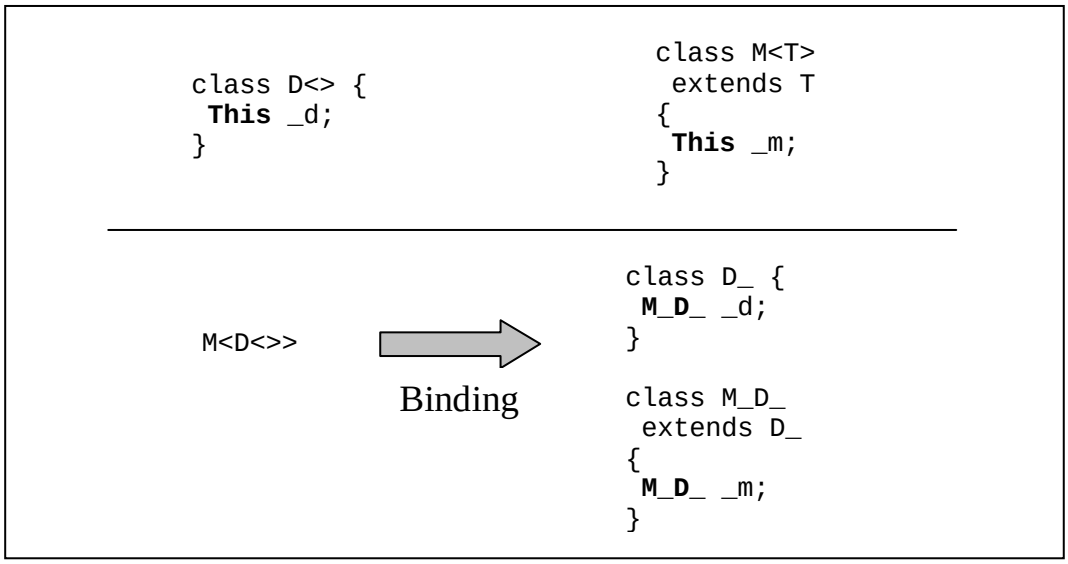


Figure 22 - Binding **This** in a Mixin Class

4.3.2 AN OBJECT-ORIENTED PROGRAMMING PROBLEM

This section describes how **This** addresses a general problem involving inheritance in object-oriented programming. Figure 23 shows two Java classes that define the nodes used in linked lists. `SingleLink` defines nodes in singly linked lists; `DoubleLink` defines nodes in doubly linked lists by extending `SingleLink`. This use of inheritance, however, leads to an asymmetry in `DoubleLink`: its `_next` field is of type `SingleLink` and its `_prev` field is of type `DoubleLink`.

```
class SingleLink {
 Object _data;
 SingleLink _next;
}

class DoubleLink extends SingleLink {
 DoubleLink _prev;
}
```

Code Fragment

```
DoubleLink d = …           // Initialize variable d
DoubleLink e = d._next();  // Compile error, needs type cast
```

Figure 23 - List Nodes in Java

`DoubleLink`'s asymmetry causes two problems. First, the `_next` field in `DoubleLink` is not precisely type checked since a `SingleLink` object could be assigned to the `_next` field in a doubly linked list. Second, manually inserted type casts are sometimes required when using the `_next` field, as the code fragment in Figure 23 illustrates.

We would like to define a `DoubleLink` class that reuses the `SingleLink` class *and* is type-safe. In JL, both objectives can be met using parameterized types and **This**, as we see in Figure 24. The top portion of the figure redefines the node classes as parametric types. The first instantiation shows that if `SingleLink<>` is used alone, then **This** is bound to `SingleLink<>`. The second instantiation shows that if `SingleLink<>` is used as a parent class, then **This** is bound to the most derived class that is generated. In the instantiation of `DoubleLink<>`, **This** is bound to `DoubleLink<>` in both generated classes to achieve the type safety that we want.

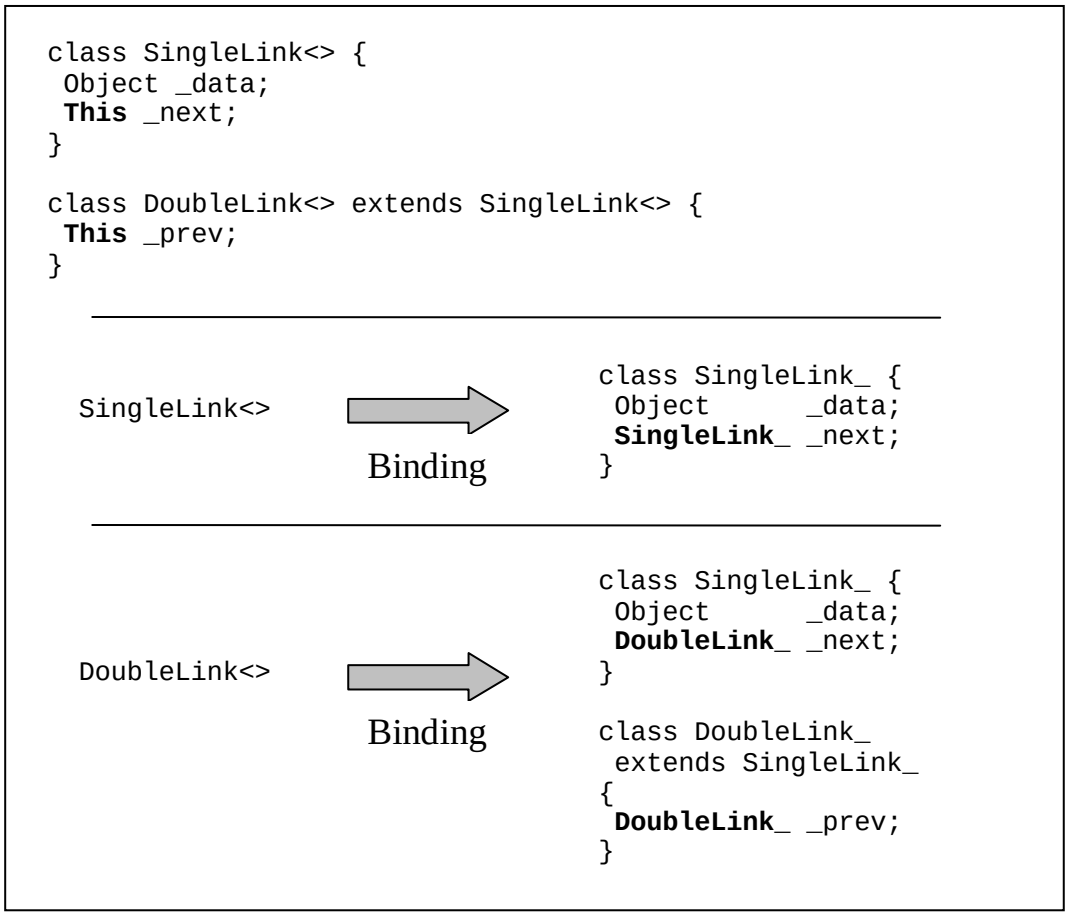


Figure 24 - List Nodes in JL

Figure 25 shows an alternative definition of list nodes in which `DoubleLink<T>` is defined as a mixin. The instantiation of `DoubleLink<T>` with `SingleLink<>` binds all occurrences of **This** to the most derived class that is generated, just as the instantiation of `DoubleLink<>` above in Figure 24 does.

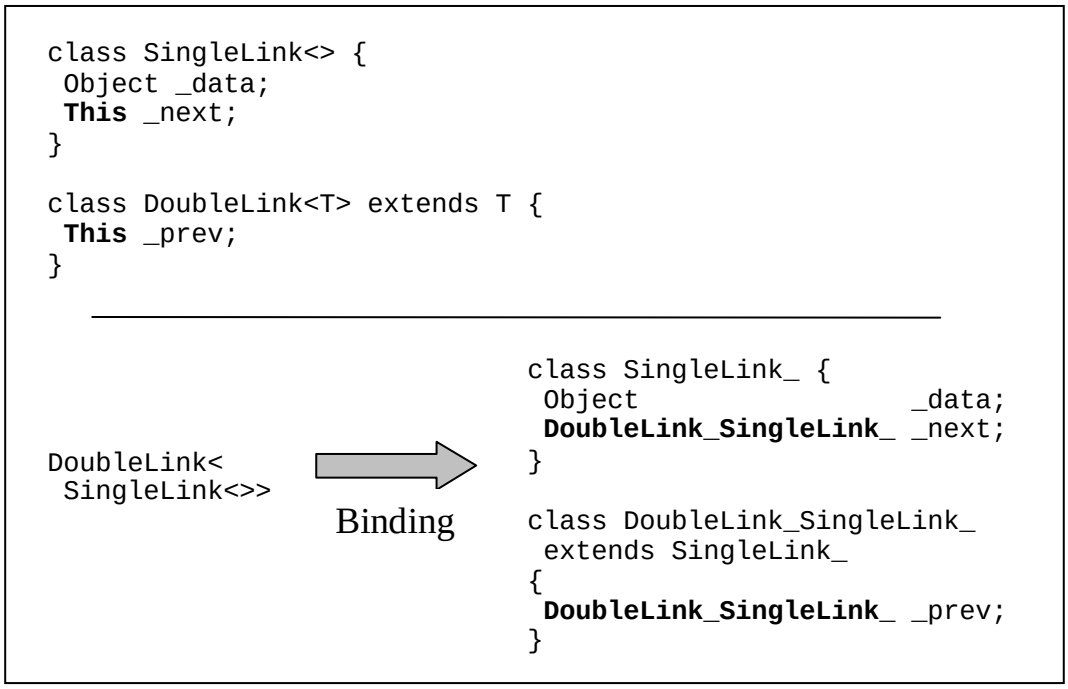


Figure 25 - List Nodes in JL Using Mixins

4.3.3 BINDING *THIS*

We described **This** usage in the previous two sections. In this section, we specify how **This** works by defining the binding process. This-binding occurs either *implicitly* or *explicitly* whenever a parametric type is instantiated. We first discuss implicit binding and then discuss explicit binding.

Implicit This-binding is sensitive to the context in which an instantiation appears. Instantiations are in an *inheritance context* when they occur (1) in an extends or implements clause of a class definition or (2) in an extends clause of an interface definition. Otherwise, instantiations are in a *non-inheritance context*. We now describe how **This** is implicitly bound in each of these contexts.

4.3.3.1 Implicit Binding in Non-Inheritance Contexts

Instantiations that appear outside inheritance clauses are said to be in non-inheritance contexts. Non-inheritance contexts include field or variable declarations; formal parameters or return types of methods; cast, instanceof or allocation expressions; type parameter constraint clauses; and non-mixed-in parameters to other type instantiations.

When a parametric type is in a non-inheritance context, then the instantiation of that type will be the most derived type in the generated hierarchy. For parametric classes, this most derived type is the class that we want to bind to **This** (§4.3.1). For parametric interfaces, no class binding for **This** is generated, so we use *Object*. Thus, the rules for implicit This-binding in non-inheritance contexts are straightforward:

- The implicit binding for a *parametric class* is the class's instantiation.
- The implicit binding for a *parametric interface* is *java.lang.Object*.

Figure 26 shows class `D`, which instantiates three parametric classes, `A`, `B` and `C`, and a parametric interface, `I` (none of which is shown). None of the parametric types are mixins, so all instantiations in `D` appear in non-inheritance contexts. Using the two rules above, **This** is bound in each instantiated type as indicated in the figure.

```
class D {
 A<int>       _a;
 A<B<Vector>> _ab;
 B<String> m(C<HashMap> map){I<HashMap> imap = map; …}
}
```

Instantiation	This-Binding
A<int>	A_int
A<B<Vector>>	A_B_Vector
B<String>	B_String
C<HashMap>	C_HashMap
I<HashMap>	java.lang.Object

Figure 26 - This-Binding in Non-Mixin Classes

4.3.3.2 Implicit Binding in Inheritance Contexts

In an inheritance context, the type being defined (the *defining subtype*) inherits from an instantiated parametric type (the *instantiated supertype*). The This-binding in the instantiated supertype depends on (1) whether the defining subtype is a class or interface and (2) whether the defining subtype is parametric or non-parametric. We now describe how each of the four possible subtype combinations supports the semantics of **This**.

If the defining subtype is a non-parameterized class, then **This** in the instantiated supertype is bound to the defining subtype itself. If the defining subtype is a parameterized class, then **This** in the instantiated supertype is bound to the This-binding of the defining subtype.

When the defining subtype is an interface, the instantiated supertype must also be an interface. If the defining subtype is a non-parameterized interface, then **This** in the instantiated supertype is bound to *java.lang.Object*. If the defining subtype is a parameterized interface, then **This** in the instantiated supertype is bound to the This-binding of the defining subtype. Table 3 summarizes implicit This-binding in inheritance contexts.

Defining Subtype	**This-Binding in Instantiated Supertype**
Non-Parametric Class	Defining subclass
Parametric Class	This-binding of defining subclass
Non-Parametric Interface	*java.lang.Object*
Parametric Interface	This-binding of defining subinterface

Table 3 - Implicit Bindings in Inheritance Contexts

To see how This-binding works, we present examples of each of the four cases described in the table above. Our examples show code fragments that instantiate types and use the simplified naming convention for instantiations described in §4.3.1.

In the code fragment below, non-parametric class `D` extends class `C<>`. **This** is bound to `D` in `C_` (the instantiation of `C`) using the non-parametric class rule.

```
class C<> {…}
class D extends C<> {…}
```

Below, class `E<>` extends class `C<>`. The declaration of field `e` binds **This** to `E_` in both `C_` and `E_`. Also shown is the declaration of field `f`, which instantiates mixin `F<T>` with `C<>`. In `F<C<>>`, **This** is bound to `F_C_` in both generated classes, `C_` and `F_C_`. The parametric class rule is used in the declarations of fields `e` and `f`.

```
class C<> {…}
class E<> extends C<> {…}
E<> e;
class F<T> extends T {…}
F<C<>> f;
```

Below, interface `J` extends interface `I<>`. **This** is bound to *java.lang.Object* in `I_` using the non-parametric interface rule.

```
interface I<> {…}
interface J extends I<> {…}
```

Below, interface `K<>` extends interface `I<>`. The declaration of field `k` binds **This** to *java.lang.Object* in both `K_` and `I_`. **This** is bound in `K_` using the interface rule for non-inheritance contexts (§4.3.3.1); **This** is bound in `I_` using the parametric interface rule. Also shown is class `D`, which binds **This** to `D` in both `K_` and `I_`. In this case, the non-parametric class rule determines the binding in `K_` and the parametric interfaces rule determines the binding in `I_`.

```
interface I<> {…}
interface K<> extends I<> {…}
K<> k;
class D implements K<> {…}
```

The simple naming scheme that we have been using is fine for illustrative purposes, but is not adequate for JL's implementation. One problem is that the same name is used for instantiations with different This-bindings. For instance, in the last example, the name `K_` is used for instantiations of `K<>` with **This** bound to *java.lang.Object* and with **This** bound to `D`. Clearly, both versions of `K_` cannot represent the same type because each version can declare different fields and methods using **This**. Since types are distinguished by name in Java, **This** must be used along with explicitly declared type parameters to distinguish instantiations.

We leave the details of JL's naming scheme until §6.3, but we note that even when **This** is used to distinguish instantiations, implicit This-binding does not always instantiate the precise type that we need. To allow for greater control of over type generation, JL provides a way to explicitly bind **This**, which we now describe.

4.3.3.3 Explicit Binding

In this section, we describe how JL generates types and how This-binding affects type generation. We then consider cases in which implicit This-binding does not generate the types we need. To address these cases, we introduce explicit This-binding, which allows programmers to override default This-binding.

In Java, types are identified by their fully-qualified names. In JL, parametric types are not Java types (§4.1); they are *type functions* or *type schemas* that

generate Java types when they are instantiated. Each distinct instantiation of a parametric type produces a distinct Java type. Therefore, only instantiations generated from the same parametric type with the same actual parameters can have the same Java type.

In JL, a parametric type name is combined with all its actual type parameter names, including the name of the type bound to **This**, to uniquely identify the type of an instantiation. This process of *name mangling* is described in detail in §6.3. For the purposes of this discussion, we now extend the simple naming scheme of the previous sections to incorporate the This-binding:

> The *name of an instantiated type* consists of the parametric type's name, the name of the This-binding, followed by the names of the actual type parameters in declaration order. We precede the This-Binding component with a colon (:) to distinguish it.

Figure 27 uses this new naming scheme to illustrate how the default This-binding is not always appropriate. Class `C` implements interface `I<>`, which binds **This** in `I<>` to `C`. In addition, class `D<T>` constrains type parameter `T` with `I<>`, which binds **This** in `I<>` to *java.lang.Object*.

The lower two portions of Figure 27 show the instantiations generated by `C` and `D<T>`. The instantiation `D<C>` (not shown) would fail because `C` does not implement the interface required by type parameter `T`. More precisely, `D<C>` would fail to compile because `C` is a subtype of `I_:C`, not `I_:Object`.

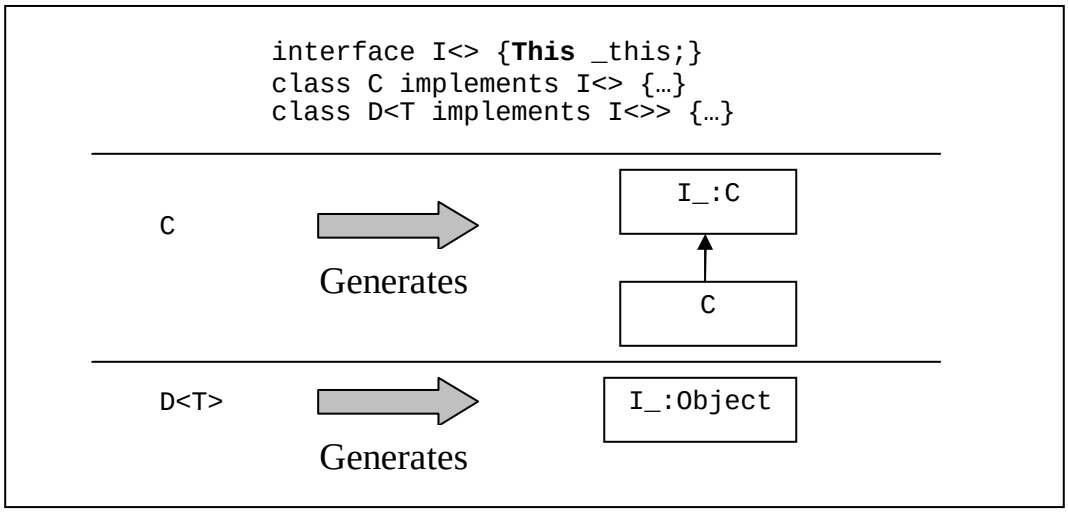


Figure 27 - The Need for Explicit This-Binding

The constraint clause in the above example uncovers an undesirable interaction between type parameter constraints and **This**. Classes that do not implement `I<>` cannot be bound to `T`, which means the constraint clause is working as expected. On the other hand, classes that *do* implement `I<>` never bind **This** in `I<>` to *java.lang.Object* (§4.3.3.2), so they too cannot be bound to `T`. Thus, no class can satisfy the constraint on type parameter `T`. This example shows how implicit This-binding prohibits certain uses of parametric interfaces in constraint clauses, even though the intended semantics are reasonable.

The restriction on constraint clause usage points to a more fundamental issue. The types generated by parametric types depend on the context in which instantiation takes place. Mixing types generated in inheritance contexts with those generated in non-inheritance contexts often leads to mismatches. For example, using the definitions from Figure 27, the code fragment below does not compile

because `C` is not a subtype of `I_:Object`, the same reason constraint checking failed above.

```
I<> _ifc = new C(); // Compile error.
```

Explicit This-binding allows programmers to control the binding of **This** in all contexts. Figure 28 redefines the constraint clause from the previous example using explicit This-binding notation. The instantiation of `D<C>` (not shown) would now compile because `T`'s constraint is satisfied. Alternatively, redefining `C` as "`class C implements I<:Object> {…}`" would also allow the original definition of `D<C>` in Figure 27 to compile.

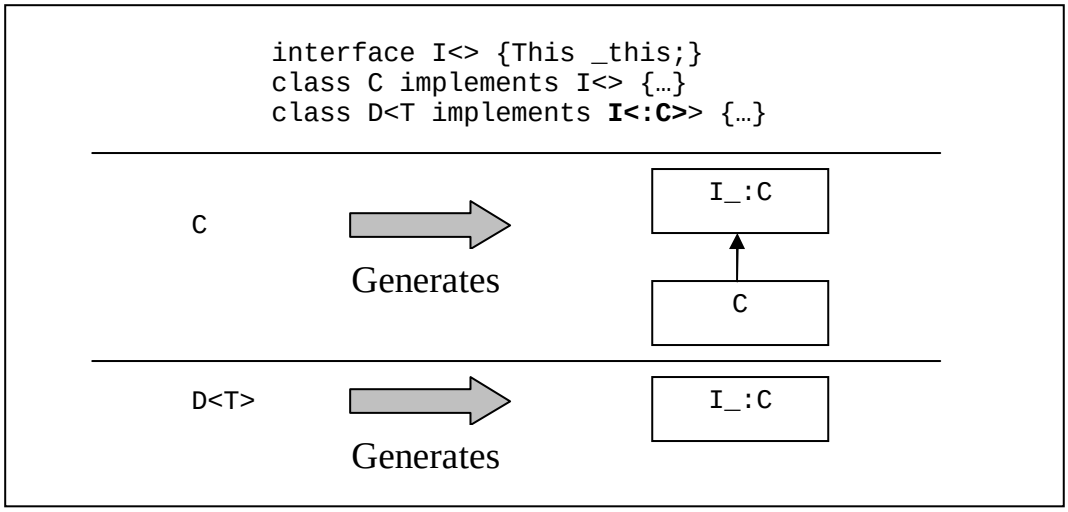


Figure 28 - Explicit This-Binding

This is explicitly bound by specifying a colon (:) followed by a class type as the first actual parameter to a parametric type. Any class can be explicitly bound in a parametric interface. Only subclasses of the parametric class being

instantiated can be explicitly bound in a parametric class.[5] This restriction preserves the notion of **This** as a class type generated by an instantiation.

In §6.3, we describe JL's naming scheme for instantiated types that ignores **This** when it is safe to do so. For an important class of instantiations, JL's naming scheme avoids the compatibility issues described in this section and the need for explicit This-binding. We now discuss This-binding in nested parametric types.

4.3.3.4 Qualified *This* Usage

This section describes JL support for *qualified* **This** names, which allows a nested parametric type access to the **This** type parameters of its enclosing parametric types. In JL, all parameterized types have their own This-binding; this is true even if one parameterized type happens to be nested within another. The scope of a type parameter includes the body of the declaring type, including any nested parametric types (§4.1.3). Generally, a nested type cannot redefine a type parameter if that type parameter is already defined in an enclosing scope. The implicit **This** type parameter, however, is the exception to this rule, so special syntax is provided to access all in-scope **This** parameters.

Figure 29 shows three parametric classes, `A<T>`, `B<U>` and `C<>`. `B` accesses the **This** parameter of `A<T>`, and `C` accesses the **This** parameters of both `A<T>` and `B<U>`. Qualified **This** names have the form *X.This*, where *X* is an identifier that represents an enclosing parametric type name. *X.This* is replaced by the This-binding of *X* during instantiation.

---

[5] As usual, *subclass* is used inclusively here to include the parametric class being instantiated.

```
class A<T> {
 class B<U> {
  A.This _athis;

  class C<> {
   A.This _athis;
   B.This _bthis;
  }
 }
}
```

Figure 29 - Qualified **This** Usage

4.3.4 DISCUSSION

In previous sections, we defined JL's implicit **This** type parameter. In this section, we discuss design and implementation topics involving **This**. We first compare **This** to explicitly declared type parameters and then describe the compile-time information flow required to implement **This**.

The implicit **This** type parameter differs from explicitly declared type parameters in two ways. First, a nested parametric type's **This** parameter hides its enclosing parametric type's **This** parameter. By contrast, explicitly declared type parameters cannot be hidden because they cannot be re-declared inside their scope. By using qualified **This** expressions (§4.3.3.4), hidden **This** parameters can be accessed throughout their scope just like explicitly declared type parameters.

The second difference between **This** and explicitly declared type parameters is that **This** has an implicit type constraint when it appears in parametric classes (§4.3.3.3). In parametric classes, **This** can only be bound to subclasses of

the instantiated class in which it appears. This constraint is automatically satisfied when **This** is implicitly bound and is checked by the compiler when **This** is explicitly bound.

Except for these two differences, **This** can be used like any other type parameter. In particular, the scope of **This** is the same as the scope of all other type parameters, which includes type parameter declaration clauses, inheritance clauses, and the bodies of types (§4.1.3).

The semantics of **This**, however, are richer than other type parameters because of the way **This** is bound. Consider, for example, class `A<T>` in Figure 30. `A<T>` contains nested class `InnerA`, which uses **This** to inherit from the most derived subclass of `A<T>`. Class `D` generates two instantiations of `A<T>`. In both instantiations, **This** is bound to `D`, which means that both `InnerA` classes inherit from `D`. Note that the semantics of `InnerA` change if it inherits from `A<T>` instead of **This**.

```
// InnerA extends the most derived subclass of A<T>.
class A<T> extends T {
 static class InnerA extends This {…}
}

// D is bound to This in both instantiations of A<T> in the
// composition below.  Both InnerA classes extend D.
class D extends A<A<LinkedList>> {…}
```

Figure 30 - The Expressiveness of **This**

Figure 31 illustrates other semantically interesting uses of **This**. Class `B<T>` contains a parametric nested class `InnerB`, which inherits from the most

derived subclass of it enclosing class, `B<T>`. Class `C<T>` defines a field type that instantiates `A<T>` with the most derived subclass of the field's declaring class, `C<T>`.

```
// InnerB<> extends the most derived subclass of B<T>.
class B<T> {
 static class InnerB<> extends B.This {…}
}

// A<T> from the previous figure is instantiated with the
// most derived type generated by the instantiation of C<T>.
class C<T> {A<This> _fld; …}
```

Figure 31 - More Ways to Use **This**

Figure 32 shows invalid uses of **This** that can be easily detected by the compiler. A parametric class that inherits from **This** or from a member of **This** would generate class hierarchies with circular inheritance dependencies. Also, interfaces cannot inherit from **This** because they cannot inherit from classes.

```
// Circular dependencies.
class D<> extends This {…}                // compile error
class E<T> extends This.Inner {…}         // compile error

// Interfaces cannot extend classes.
interface I<T> {
 interface InnerI<> extends I.This {…}    // compile error
}
```

Figure 32 - Invalid **This** Usage

The last discussion topic concerns the implementation of **This**. The challenge in implementing **This** is managing information that flows in both directions

in mixin-generated hierarchies. This information flow is needed to name instantiations using the naming convention described in §4.3.3.3. At instantiation-time, This-binding information flows from leaf to root and is used to name supertypes. At the same time, type information flows from root to leaf and is used to name subtypes. Figure 33 depicts this two-way information flow. The precise way instantiations are named in JL is described in §6.3.

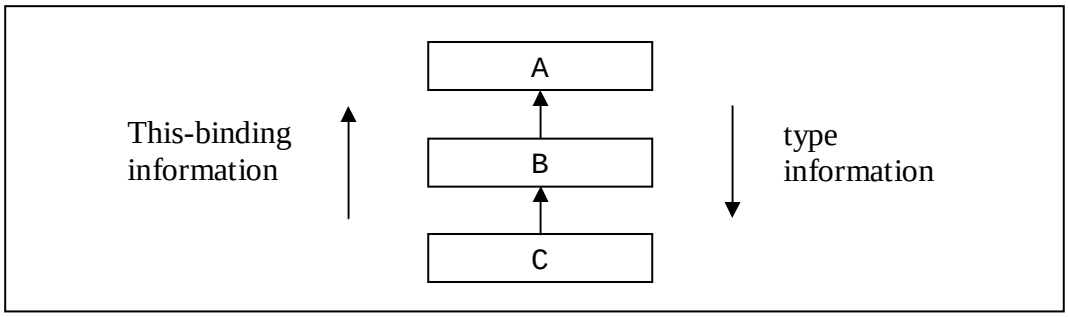


Figure 33 - Two-Way Information Flow at Instantiation-Time

Before concluding our discussion, we note that **This** is an important component of the Sibling design pattern, which we discuss in §7.2.3.4. We now discuss work related to **This** and the notion of most derived type.

4.3.5 RELATED WORK

Thorup proposes to add genericity to Java using *virtual types* [120], which are types that automatically adapt when they are subtyped [67,74]. Using virtual types, a child class that inherits from a parent class can cause types declared in the parent to change and become more specialized. Thorup's implementation of virtual types for Java solves the object-oriented programming problem described in the linked list example in §4.3.2. The implementation also allows the type of *this*

to be expressed, which means that the most derived type in any class hierarchy can be specified.

JL's implicit **This** type parameter can be seen as a limited, static, virtual typing mechanism. JL combines the power of mixins, which are not present in Thorup's proposal, with an expression of the most derived type generated by an instantiation of parametric types. In JL, **This** can only be used in parametric types. In Thorup's proposal, virtual types bring a full range of generic capabilities to all Java types, including the ability to express the most derived type. This expressiveness, however, comes at the cost of increased dynamic type checking. JL, on the other hand, avoids runtime overhead by processing **This** statically at instantiation-time.

Bruce and colleagues [26,27] propose a parametrically polymorphic implementation of Java with some virtual typing capabilities. Their proposal augments Generic Java[6] [24] with *ThisType* and an exact type operator. *ThisType* is defined as the public interface of *this*. Support for *ThisType* breaks the equivalence between subtyping and subclassing in Java. This break requires the introduction of *matching* for type checking, which requires a new programming model in which subclasses match their superclasses but subclasses do not always subtype their superclasses.

JL, on the other hand, does not change Java's type system nor does it change the semantic relationship between subclassing and subtyping. In JL, **This** refers to a class type, which often does not correspond to the public interface of

[6] See §4.1.4.1 for a discussion of Generic Java.

this. Bruce points out two pitfalls to using class types as JL does. First, interfaces used as standalone types have no class type, so **This** cannot be bound. Second, calls to methods requiring an actual parameter of type **This** cannot be statically type checked because the type of the receiving object is not generally known at compile time. JL avoids the first problem by implicitly binding **This** in all situations, including in interfaces, when no explicit binding is given. JL avoids the second problem by limiting the use of **This** to parameterized types, which allows **This** to be statically bound at instantiation-time.

Czarnecki and Eisenecker [36] can also reference the most derived type generated by C++ template classes. Their approach involves the use of a separate, manually configured repository class that is passed as a type parameter to template classes during instantiation. The type of the instantiation itself is specified in the configuration class. As in JL, all processing takes place statically. The most derived type can be referenced from within the template instantiation, but some of the type's members cannot be accessed due to the way C++ processes templates.

A form of virtual typing has also been implemented in domain-specific GenVoca [15] generators. In these generators, the types that can be refined are known in advance, and this knowledge is typically built into the domain-specific compiler. In some GenVoca implementations, such as P3 [14], programmers can express the most derived type generated by a composition of GenVoca components. Here, the implementation of most derived type is distinct from the mecha-

nism of component parameterization; in JL, the implementation extends the mechanism of component parameterization.

4.4 Constructor Propagation

Constructor propagation is a new dynamic approach to superclass initialization developed for Java Layers. The main goal of this approach, especially when compared to the static approaches described in §2.4.1, is to enhance the flexibility and reusability of mixins. The constructor propagation algorithm described in §4.4.2 specifies how subclass constructors are adapted to their superclass constructors. The contribution of this algorithm is that the problem of superclass initialization for mixins (§2.4.1) is solved with the addition of a single keyword to the language and with an implementation that is orthogonal to other JL feature implementations. We introduce constructor propagation in this section; we describe its algorithm, its usage, and related work in the following sections.

In Java Layers, programmers use the new **propagate** modifier to designate non-private constructors as participants in the constructor propagation algorithm. Constructors designated in this way, as well as the default constructor, are called *propagatable*.[7]

Constructors are propagated from parent to child class, with propagatable constructors in the parent only able to affect propagatable constructors in the child. In the simplest case, when the child has no explicitly declared constructors, constructor propagation acts like constructor inheritance and replicates appropri-

[7] JL's **propagate** modifier is used for both constructor propagation and deep conformance (§4.2).

ately modified parent constructors in the child. In more complex cases, constructor propagation changes the signatures and bodies of existing child class constructors. These changes allow the child class constructors to automatically pass arguments to each of the propagatable parent class constructors.

Figure 34 shows the signatures of the propagatable constructors in parent class *P* and its child class *C*. During constructor propagation, the signatures of the *C*'s two constructors are changed to take the `String` parameter from *P*'s constructor. Though not shown, *C*'s constructors are also modified to explicitly invoke *P*'s constructor with a `String` argument.

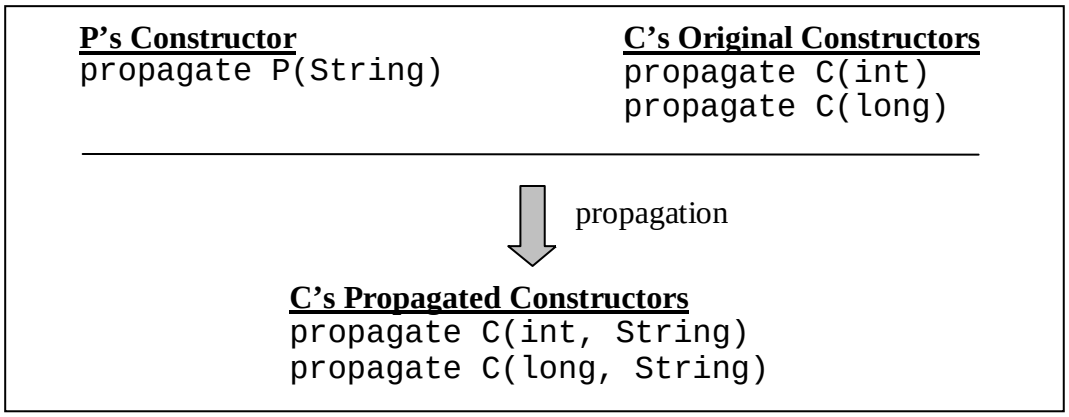


Figure 34 - Constructor Signatures in Parent and Child

In parametric types, constructor propagation is performed during instantiation. This instantiation-time processing allows a mixin class's constructors to be generated using its superclass's constructors, which are now known. In a mixin hierarchy, constructor propagation automatically adjusts constructor signatures so that all classes in the hierarchy can be properly initialized.

Both parametric and non-parametric classes can contain propagatable constructors. JL's current support, however, extends only to parametric classes who source code is being compiled. Full support for constructor propagation requires adding the new **propagate** attribute to the class file representation of constructors. This type of extension was anticipated by Java's architects and can be easily implemented in an upwardly compatible way (JVMS §4).

Constructor propagation does not depend on any other JL language feature and could be integrated into Java as a standalone feature, though the need for doing so in a language without mixins is arguably small. We now describe the constructor propagation algorithm and illustrate its usage.

4.4.2 ALGORITHM

Informally, the propagation algorithm takes the formal parameters of each propagatable superclass constructor and appends them to the signatures of copies of all propagatable subclass constructors. In this way, the subclass's original propagatable constructors are used as prototypes for constructor generation and are then discarded after the algorithm is applied.

More precisely, for parent class *P* and its direct child class *C*, the propagation algorithm performs the following:

1. **for each** propagatable constructor *p* originally in *P*
2. **for each** propagatable constructor *c* originally in *C*
3. Create copy *c'* of *c*
4. Append *p*'s parameters to *c'*, mangling names when necessary
5. Insert a call to *p* as the first statement in *c'*
6. Add *c'* to *C*
7. Discard all of *C*'s original propagatable constructors

On lines 1 and 2 above, a nested loop processes each propagatable constructor in *P* with each propagatable constructor in *C*. We process only those propagatable constructors that exist in each class at the start of the algorithm. On line 3, *C*'s current constructor is copied to *c'*. On line 4, the signature of *c'* is augmented with the formal parameters of *P*'s current constructor, *p*. Each formal parameter from *p*, in left to right order, is appended to *c'*, with the parameter name mangled if a name conflict would otherwise occur. On line 5, a call to *p* is inserted into *c'*, allowing *p* to obtain the parameters it needs from *c'*. On line 6, *c'* is inserted into *C*. After all propagatable constructors are processed, *C*'s original propagatable constructors are deleted (line 7).

In Java, explicit constructor calls (JLS §8.8.5.1) use the **super** or **this** keywords with an argument list. Explicit constructor calls can appear only in the first statement of a constructor body. JL reports a compilation error if a constructor designated as propagatable already contains an explicit constructor call. This restriction guarantees that line 5 in the algorithm generates valid constructor code, which can contain at most one explicit constructor call.

The restriction on explicit constructor calls can be relaxed to increase JL's flexibility: If a propagatable constructor contains an explicit call to another propagatable constructor in the same class (i.e., the explicit call uses **this**), then valid code can still be generated. This relaxed restriction simply allows superclass constructor parameters to pass through some number of intermediary subclass constructors before arriving at the superclass.

4.4.3 DISCUSSION

In this section, we describe the behavior of the constructor propagation algorithm defined in §4.4.2. Figure 35 defines four simple classes, three of which have explicitly declared propagatable constructors. These classes contain only the constructors shown. Table 4 lists four instantiations using these classes and the constructors that are generated.

```
class A                          class B
{                                {
 propagate A(TypeA a){…}          propagate B(){…}
}                                }

class M<T> extends T             class N<T> extends T
{                                {}
 propagate M(TypeM m){…}
}
```

Figure 35 - Classes with Propagatable Constructors

Instantiations	Signatures of Generated Constructors
N<A>	N_A(TypeA a)
M<B>	M_B(TypeM m)
M<A>	M_A(TypeM m, TypeA a)
M<M<A>>	M_A(TypeM m, TypeA a), M_M_A(TypeM m, TypeM m1, TypeA a)

Table 4 - Generated Constructors

The instantiations in Table 4 generate classes with names derived from the names of the composed classes. These derived class names are reflected in the

names of the constructors in the right-hand column of the table.[8] All generated constructors are themselves propagatable as line 3 in the constructor propagation algorithm implies (§4.4.2). We now describe each of the four example instantiations in detail.

The first instantiation, `N<A>`, shows how constructors are generated in mixins that declare no explicit constructors. In this case, superclass `A`'s constructor is propagated with the default constructor in `N`. The result is that the constructor in instantiated class `N_A` simply forwards its parameter to `A`'s constructor.

The second instantiation, `M<B>`, shows how no-argument superclass constructors leave subclass constructors unchanged. In this case, the no-argument constructor in `B` does not propagate any parameters, so the constructor in instantiated class `M_B` takes the same parameters as `M`'s constructor. The outcome would be the same if `B` used the default constructor.

The third instantiation, `M<A>`, shows how the signatures of two propagatable constructors are combined when both constructors have formal parameters. In this case, superclass `A`'s constructor is propagated with the constructor from `M`. The result is that the constructor in instantiated class `M_A` takes the parameter from `M`'s constructor (`m`) followed by the parameter from superclass `A`'s constructor (`a`).

The fourth instantiation, `M<M<A>>`, shows how constructor propagation accumulates parameters in a class hierarchy. In this case, the constructor for instantiated class `M_A` is first generated as it was in the previous example. Next,

[8] The generated names are simplified versions of the names JL actually generates; see §6.3.

`M_A`'s generated constructor becomes the superclass constructor that is propagated with `M`'s constructor. The result is that the constructor in instantiated class `M_M_A` takes the parameter from `M`'s constructor (`m`) followed by the parameters from superclass `M_A`'s constructor (`m` renamed to `m1` and `a`).

In mixin-generated hierarchies, constructor propagation copies formal parameters from root class constructors to leaf class constructors. Glue code is then inserted to forward actual parameters from leaf to root during execution. In layered applications, mixin classes often implement features that require no initialization; constructors in these mixins exist only to forward parameters to superclass constructors. In the examples above, we saw how constructor propagation automatically generates forwarding constructors and thereby reduces the work for programmers. In our Fidget evaluation (§7.2), we show that the benefit from this reduction can be significant.

Constructor propagation, however, must be used with some restraint because the number of generated constructors can grow geometrically. Consider, for example, a hierarchy of three classes in which the root class contains 3 propagatable constructors and each of the other classes contains 4 propagatable constructors. The total number of constructors generated in the leaf class alone is (3)(4)(4) = 48. This multiplicative effect has the potential to create a large number of constructors, many of which will probably never be used. Despite this potential for abuse, conservative use of constructor propagation is both possible and effective (Chapter 7). In practice, a compiler can limit the number of constructors generated or allow the user to selectively reduce the set of generated constructors.

4.4.4 RELATED WORK

In this section, we relate constructor propagation to other approaches to superclass initialization in mixin programming. First, we describe the predominant use of static approaches in existing work. Next, we describe an approach that is still static, but increases flexibility by using some dynamic features. Finally, we compare constructor propagation with a dynamic approach to superclass initialization that uses C++ templates [110].

4.4.4.1 Static Approaches

Static approaches to superclass initialization assume fixed signatures of superclass constructors. In §2.4.1, we described how this assumption limits the reusability and flexibility of mixins. Despite these limitations, static superclass initialization is the most common approach used in mixin programming because of its simplicity and, we argue, the lack of language support for alternative approaches. We note that static approaches and constructor propagation can be used in the same application.

VanHilst and Nokin [126] recognized the superclass initialization problem in their graph traversal application, but the application's small number of mixins could easily be managed with static initialization. The same researchers also used mixins to rebuild a text sorting program [125] and a recycling machine program [127]. Here, the number of collaborating objects was fixed and still relatively small, so superclass initialization could again be handled statically.

The superclass initialization problem has been recognized in a number of GenVoca-related studies, including one that focused on mixin programming in

C++ [104] and another that defined the initial version of Java Layers [30]. In the latter report, a preliminary version of constructor propagation was designed and implemented (§6.1).

Static superclass initialization is used in all GenVoca-related research prior to Java Layers. This category includes P++ [100], which is a domain-independent language extension to C++ that supports the GenVoca model. This category also includes the domain-specific P3 [11] extension to Java and the Jark-arta Tool Suite [14], which is used to implement P3.

4.4.4.1 A More Flexible Static Approach

Static superclass initialization as described in §2.4.1 can be made more flexible without using a fully dynamic approach like constructor propagation. One alternative, for example, uses an *argument class* and a programming convention that requires all constructors in an application to define the argument class as their only formal parameter. This argument class approach is static because constructor signatures are fixed, but mixin compositions do not fail because a single, well-known constructor interface is supported by all classes.

Eisenecker, Blinn and Czarnecki [38] define argument classes that store initialization data in fields. By convention, classes access their initialization data from pre-assigned fields in the argument class. Under this approach, the argument class is updated whenever any application class requires new initialization data.

For even greater flexibility, initialization data can be kept in a hash table in the argument class. This makes the argument class insensitive to changes in

initialization data. Under this design, programmers establish a key naming convention for table entries so that classes can appropriately access their initialization data. Variations on this design can use data structures other than hash tables for the argument class.

When argument classes are used, constructors do not reflect the precise interface used to initialize classes. Instead, users learn through documentation or some other convention what data is needed for the different classes; runtime errors occur if the wrong data is supplied. On the other hand, when constructor propagation is used, constructors reflect the precise interfaces used to each initialize classes and runtime errors are avoided through static type checking.

4.4.4.2 A Dynamic Approach Using C++ Templates

Eisenecker, Blinn and Czarnecki [38] define a dynamic approach to superclass initialization for C++ mixins [36,104,110]. This approach extends the idea of argument classes described in the previous section to that of automatically generated lists of arguments. This *argument list* approach is dynamic because the argument types passed to constructors are automatically customized for each mixin instantiation.

Central to the argument list approach is the use of heterogeneous value lists [54], which are lists generated at instantiation-time that contain elements of different types. These elements contain fields that are assigned values at runtime. In particular, argument lists are defined using the C++ types shown in Figure 36. Argument lists are variable length, singly linked lists of heterogeneous elements that are constructed by instantiating `Param`.

```
struct NIL {};

template<class T, class Next_ = NIL>
struct Param
{
 Param(const T& t_, const Next_& n_ =NIL()): t(t_),n(n_)
 {}

 const T& t;
 Next_ n;
 typedef Next_ N;
}
```

Figure 36 - Statically Generated Parameter Lists in C++

Figure 37 shows how an argument list containing a string and an integer is constructed. `Param` is instantiated twice to create a list that contains a string value followed by an integer value. The parameters of the constructors called during the creation of object `p` are statically type checked. The output statement shows how values in the argument list are accessed; the output statement prints the text shown in the figure.

```
int main
{
 Param<char*, Param<int> > p(“Madeline”, Param<int>(8));

 cout << p.t << “ is ” << p.n.t << endl;
}
```

Printed text: Madeline is 8

Figure 37 - Example Argument List

Eisenecker et al. use argument lists as parameters to mixin class constructors. Each mixin class constructor takes a customized argument list as its only parameter. In Figure 38, the authors define the `Customer` class's constructor, which takes an argument list of type `ParamType`. An interesting feature of the `Customer` class is that it uses the types defined in the type parameter `Config_` to build the argument list for its constructor.[9] This use of an auxiliary class brings us to the next level of design in the argument list approach.

[9] The C++ **typename** keyword indicates that a type parameter member is a type definition.

```
template<class Config_>
class Customer
{
 public:
  typedef Config_ Config;
  typedef Param< typename Config::LastnameType,
            Param< typename Config::FirstnameType > >
    ParamType;

  Customer(const ParamType& p)
   :lastname_(p.t),firstname_(p.n.t)
  {}

 private:
  typename Config::FirstNameType firstname_;
  typename Config::LastNameType  lastname_;
}
```

Figure 38 - Constructor with Argument List Parameter

Eisenecker et al. use a number of generative programming techniques and concepts [36] to make the use of argument lists as transparent as possible to the user. The first technique used defines a *configuration repository class* for each distinct mixin instantiation. Configuration repositories define the types of the initialization parameters stored in the argument lists of all mixin constructors in an instantiation. These repositories also contain the type of the generated leaf class of the hierarchy. The types defined in the configuration repository classes are used as type parameters to `Param` to create argument lists. In Figure 38 above, the `Config_` type parameter represents a configuration repository.

The second generative programming technique uses a generic *parameter adapter* to generate a constructor for the leaf class of a mixin-generated hierarchy. The purpose of this adapter is to generate a conventional constructor that takes

individual parameters, which are then used to build an argument list. This conventional constructor passes the argument list it constructs to its superclass constructor. The parameter adapter is implemented as a mixin class, which becomes the new leaf class in any mixin composition in which it appears.

The third generative programming technique uses a *configuration generator* to automate the creation of configuration repositories. Configuration generators allow users to avoid creating configuration repositories by hand. Configuration generators are meta-programs that produce mixin compositions, which are then instantiated. The main idea is that a user selects the features required by an application and the configuration generator uses that selection to determine which mixins should be instantiated. Each application requires its own configuration generator program.

The argument list approach does provide a dynamic solution to the superclass initialization problem for mixins, but its flexibility comes at a cost that often reduces flexibility in other areas of mixin programming. For example, by requiring the parameter adapter mixin to always be the leaf class in a mixin composition, programmers are prevented from specifying other classes as the leaf. This restriction prevents programmers from using the special properties of leaf classes for their own purposes. These properties include that the leaf class defines an instantiation's ultimate programming interface and that the leaf class is a subtype of all other mixins in the instantiation.

More significantly, however, is the loss of flexibility due to the use of configuration repository classes or configuration generators. When configuration re-

positories are used, instantiating a new combination of mixins requires the definition of a new configuration repository class, assuming the mixin constructors have parameters. In addition, changes to the parameters of existing constructors force changes to one or more configuration repositories. When configuration generators are used, creating new instantiations and changing constructor parameters require coordinated changes to the configuration generator program.

Eisenecker et al. argue for the use of configuration generators, in part, to avoid the maintenance overhead associated with configuration repository classes. Configuration generators can provide a number of other capabilities to support mixin programming, such as the detection of invalid combinations of mixins, but they do represent auxiliary code whose maintenance is tightly coupled with an application's mixins classes.

JL's constructor propagation, on the other hand, does not require the definition or maintenance of any auxiliary data structures or programs. It does, however, require the addition of a new modifier to the language. Constructor propagation can be thought of as a special purpose meta-programming mechanism. It is possible that this mechanism could be subsumed by a more general meta-programming approach, which would be an interesting research topic for future study.

4.5 Semantic Checking

This section describes the design of JL's *semantic checking* facility. In §2.4.3, we described how the presence, absence, ordering, and number of times a

mixin appears in a composition can affect program behavior. We also described how even a small number of composable mixins can be permuted in a large number of ways—possibly an infinite number of ways if duplicates are allowed.

To emphasize this scalability issue, consider an application that contains one hundred mixins that implement the same interface. These mixins represent all the features that can be composed with one another to build a full-featured type. How do programmers know which mixin combinations to use and which combinations to avoid? Type checking is of limited value because we assume that the mixins have compatible interfaces. Programmers can resort only to documentation or the code itself, both of which require a significant investment on the part of the programmer and the development organization.

The goal of semantic checking is to simplify application construction by prohibiting syntactically valid but semantically undesirable mixin combinations. Programmers can use semantic checking to explicitly restrict how classes are combined. These restrictions are expressed as pre-conditions and post-conditions on the use of inheritance and the use of type parameters. These conditions are difficult or impossible to express using type checking alone. JL's semantic checking provides a simple, flexible way for programmers to specify to the compiler the class compositions that are known to be invalid.

Semantic checking does not, however, guarantee compositional correctness, much less program correctness. Semantic checking only enforces composition constraints that programmers manually encode into class definitions. In addition, our contribution does not include an implementation of semantic checking.

We have, however, manually applied semantic checking in our ACE evaluation (§7.1.5).

We begin by presenting the model that underlies JL's semantic checking facility. We then describe how semantic attributes are added to classes and how rules are created to check those attributes. We conclude with a discussion of our design and related work.

4.5.1 THE SEMANTIC MODEL

JL's semantic checking facility models two key concepts. The first concept is that of a semantic *attribute*, which is an identifier or tag chosen by a programmer to represent a meaningful characteristic in an application. For example, an application can define a *thread_safe* attribute to indicate that certain methods are thread-safe, or a *secure* attribute to indicate that a communication protocol uses encryption. The meaning of an attribute is established by programming convention; this meaning is only valid among classes that cooperate and adhere to the established convention.

The second key concept is the *ordered attribute list*, which defines a space in which classes can express their semantic characteristics and test their semantic constraints. At compile-time, each class hierarchy is associated with its own list of totally-ordered semantic attributes. Attributes are added to lists using **provides** clauses in class definitions. Attribute lists are tested using **requires** clauses in class definitions. For convenience, we sometimes say the "attribute list of a class" instead of saying "the attribute list of the hierarchy of a class."

Figure 39 shows an instantiation of parametric classes, the two hierarchies it generates, and the attribute list associated with each hierarchy. We will use this instantiation as a running example to describe semantic checking. The instantiation uses mixins classes and non-parametric classes. Mixin class `B` takes two type parameters, the first of which binds to `B`'s superclass. We use the name of the leaf class in a hierarchy to name the hierarchy's attribute list.

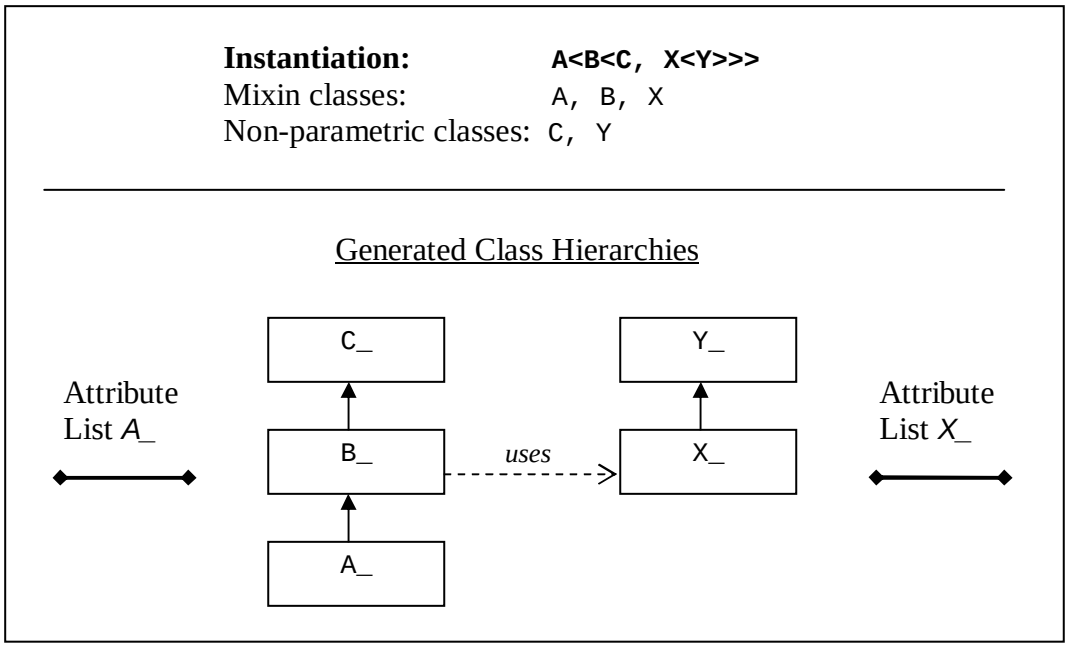


Figure 39 - Attribute Lists of Mixin-Generated Hierarchies

Figure 40 shows how attribute lists are populated. The top portion of the figure shows the **provides** clauses of the classes used in instantiation `A<B<C,X<Y>>`. The bottom portion of the figure shows how attributes from each class are placed in their associated list in declaration order. The lists are filled from left to right, starting with leaf classes and proceeding to root classes.

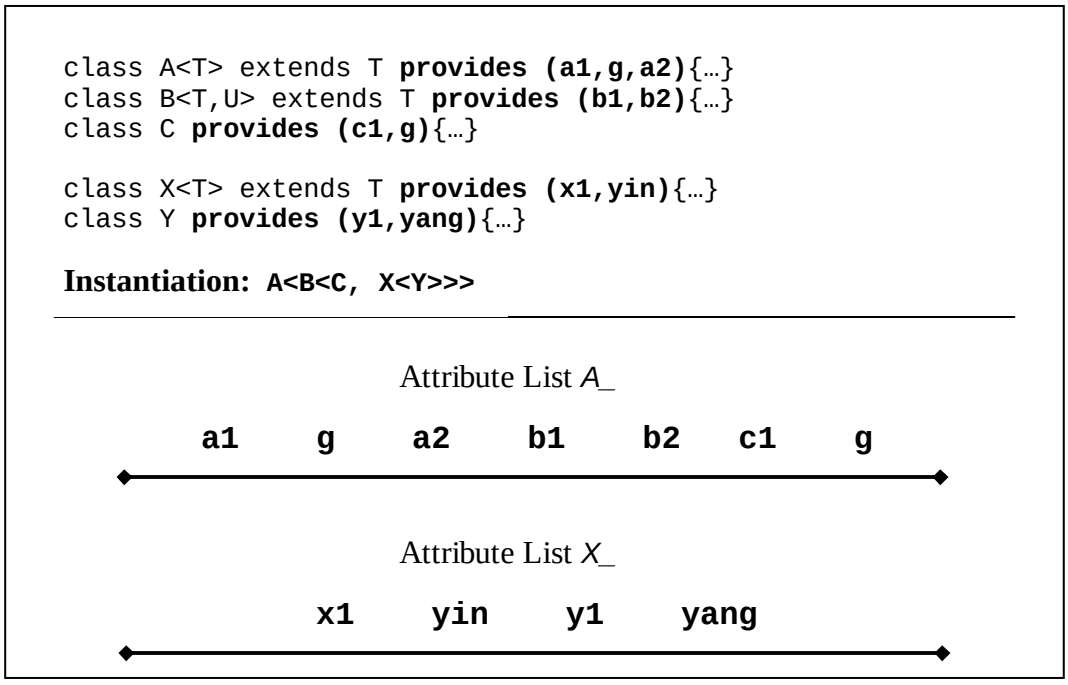


Figure 40 - Populating Attribute Lists

Attribute tests, which are also specified in class definitions, are evaluated in one of three *attribute list contexts*. The *global context* of an attribute list is the whole list, which consists of all the attributes declared in all classes of the associated hierarchy. The *superclass context* of an attribute list is defined in relation to a class *C* in which a test is specified. *C*'s superclass context is the sublist that contains only those attributes declared in *C*'s superclasses. Likewise, *C*'s *subclass context* is the sublist that contains only those attributes declared in *C*'s subclasses. Figure 41 illustrates the attribute list contexts for mixin `B` defined above.

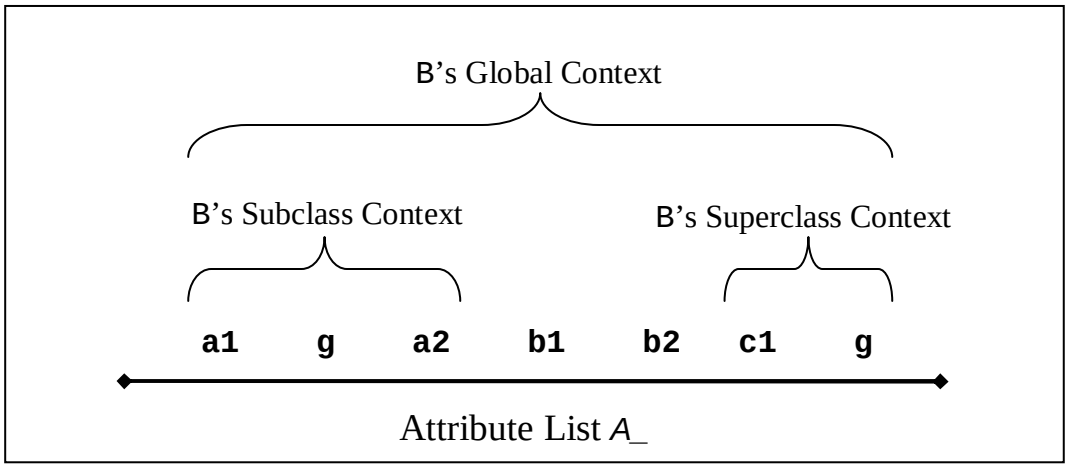


Figure 41 - Attribute List Contexts

We have described how attribute lists are created and partitioned; we now describe how they are used to test compositional semantics.

4.5.2 EXPRESSING SEMANTIC CONSTRAINTS

JL's semantic checking facility enforces constraints on ordered attribute lists. The one-to-one relationship between attribute lists and class hierarchies means that by constraining attribute lists, we can constrain the creation of class hierarchies.

Semantic checking takes place when non-parametric classes are compiled and when parametric classes are instantiated. Programmers define attributes and constraints in class definitions. The semantic checking facility generates attribute lists for all class hierarchies as described in the previous section. These lists are then tested to see if they conform to all their constraints. If an attribute list fails to satisfy any constraint, then compilation aborts with an error message.

Classes define constraints using **requires** clauses, which read but do not alter attribute lists. The constraint language uses regular expression pattern matching [42,69] and a count operator to test for the presence, absence, ordering, and cardinality of attributes in the target list context. A class can define zero or more **requires** clauses using the following grammar, which we simplify for presentation purposes: [10]

RequiresClause :: **requires** *RequiresQualifier*? *ConstraintBlock*?
RequiresQualifier :: **sub** | **super** | **unique** | *Identifier*
ConstraintBlock :: "{" *Constraint* (";" *Constraint*)* "}"
Constraint :: *RegexConstraint* | *RelationalConstraint*

In the above grammar, a question mark (?) means zero or one occurrences and an asterisk (*) means zero or more occurrences. Reserved words are in bold font and non-terminals are in italics. Double quotes and parentheses are meta-characters used to specify lexical tokens and term grouping, respectively.

The *RequiresQualifier* term specifies the context in which constraints are applied to the class's associated attribute list. If a **requires** clause has no *RequiresQualifier*, then constraints are applied in the global context. If the **sub** qualifier is used, then constraints are applied in the subclass context. If the **super** qualifier is used, then constraints are applied in the superclass context. If an *Identifier* is used, then the identifier must name a type parameter declared in the class. In this case, constraints are not applied to the attribute list associate with the class,

[10] The grammar for compound constraint expressions is not shown. Compound constraints can contain regular expressions, relational expressions, parentheses, conjunction, and disjunction.

but instead they are applied to the (whole) attribute list of the class bound to the named type parameter.

The **unique** qualifier specifies that the declaring class can appear only once in a hierarchy. This qualifier is shorthand for an idiom that uses both a **provides** and a **requires** clause to achieve the same effect. The **unique** qualifier is the only qualifier that does not have to be followed by a *ConstraintBlock* in a *RequiresClause*. If a *ConstraintBlock* does follow **unique**, however, then its constraints are applied in the global context.

A *ConstraintBlock* contains a non-empty list of constraint expressions, which are either regular expressions or relational expressions. Each expression evaluates to true or false; expressions that evaluate to false indicate that a constraint has failed, which aborts compilation. Because of the well-understood and conventional nature of regular and relational expressions, we leave the specification of their grammar as an implementation task. Instead, we now describe constraint expressions and how they are used.

Regular expressions are used to match patterns in attribute lists. Each expression specifies a pattern of attributes, which causes JL to search the target attribute list context for a match. Regular expressions check attributes for their presence, absence and ordering in list contexts.

Figure 42 shows the four **requires** clauses in mixin `B`'s definition. All constraint clauses evaluate to true in instantiation `A<B<C,X<Y>>` (Figure 40). The first clause requires that the subclasses of `B` supply both the `a1` and `a2` attributes. The second clause requires that the superclasses of `B` supply either (1) `c1` or

(2) c3 followed by c4, with zero or more intervening attributes. The third constraint clause allows only one instance of B to appear in a hierarchy. This clause also requires that some class in that hierarchy supply attribute g. The last constraint clause is evaluated in the global context of attribute list *X_* (Figure 40), which is the list associated with the class bound to U. This constraint requires that attribute z not appear in the list.

```
class B<T,U> extends T
 provides (b1,b2)
 requires sub {a1; a2}
 requires super {c1 | c3*c4}
 requires unique {g}
 requires U {!z}
{…}
```

Figure 42 - Regular Expression Constraint Clauses

The above example illustrates that regular expressions in JL use a conventional syntax [42]. JL's regular expressions are similar to those in common tools [41], such as grep, sed, awk, and emacs, and in common languages, such as Perl [130] and Python [73]. Appendix B describes JL's pattern matching syntax in more detail.

In JL, constraints are also specified using relational expressions, which enforce cardinality constraints among attributes. In addition to a relational operator (<, >, ==, !=, <=, >=), relational expressions contain integers, the addition operator (+), the subtraction operator (-), and the *attribute count operator* (#). The count operator returns the number of times an attribute appears in the target context.

Figure 43 shows how relational expressions extend the capabilities of regular expression pattern matching. Mixin `X` defines two constraint clauses, both of which use the count operator and evaluate to true in instantiation `A<B<C,X<Y>>` (Figure 40). The count operator immediately precedes an attribute, which is its parameter. The first constraint requires that the number of `yang` attributes exceed the number of `yin` attributes by one in the superclass context. The second constraint states that the number of `yin` attributes must equal the number of `yang` attributes in the global context.

```
class X<T> extends T
 provides (x1,yin)
 requires super {#yang == #yin + 1}  // Matched pair idiom
 requires {#yin == #yang}            // uses two clauses
 {…}
```

Figure 43 - Relational Expression Constraint Clauses

Together, the above two constraints specify the invariant that only matched pairs of `yin-yang` attributes exist whenever mixin `X` is used. This ability to count attributes allows the coordinated use of mixins in programs. For example, if mixin *J* acquires a resource, such as a lock or file handle, and mixin *K* releases that resource, then we can use relational expression constraints to guarantee proper *J-K* pairing in generated hierarchies.

4.5.3 DISCUSSION

JL's semantic checking facility uses a simple model in which classes advertise their properties by inserting names into an ordered list. Classes test these

lists for the properties they need and, if a test fails, compilation also fails. The goal of semantic checking is to invalidate class compositions that are known to be undesirable. For mixin class instantiations, semantic checking is especially useful because restrictions on the ordering and on the repetitive use of mixins can be conveniently expressed, which is not the case using conventional type systems.

The semantic checking facility is orthogonal to Java and to JL's other language features. Semantic checking executes statically when class hierarchies are defined and has no effect on code that executes at runtime. A class's **provides** and **requires** clauses define a sublanguage that can raise compilation errors, but is otherwise completely independent of all other language processing.

The most important characteristic of JL's semantic checking is its *simplicity*. One simplifying factor is that JL models the semantic characteristics of classes in single inheritance hierarchies, which leads to a separate (linear) list of attributes for each hierarchy. More complex approaches to semantic checking can model the semantic characteristics of tree structures, such as trees that represent multiple inheritance hierarchies. These tree modeling approaches are complicated because attribute streams need to be merged at tree nodes. In the next section, we describe a GenVoca implementation of semantic checking that merges multiple attribute streams.

Another simplifying factor in JL's semantic checking is that constraints are expressed using only regular expression pattern matching and attribute counting. This minimalistic constraint language is based on well-understood theory and on the common usage of regular expressions in programming. Most pro-

grammers are familiar with regular expressions from the file systems, tools and languages that they already use [42], so semantic checking in JL requires little training.

To increase usability, we recommend that implementations of semantic checking support meaningful error messages. For example, programmers should be able to annotate constraint definitions with error messages that are printed whenever a constraint fails. Figure 44 shows simple constraint annotations using string literals; other more elaborate approaches are also possible.

```
class X<T> extends T
 provides (x1,yin)
 requires super {#yang == #yin + 1 "Need one unpaired yang"}
 requires {#yin == #yang "The cosmos is out of balance"}
 {…}
```

Figure 44 - Programmer Defined Messages in Constraint Checking

4.5.4 RELATED WORK

A number of ad-hoc methods have been used to restrict how parametric types are instantiated. These methods rely on side-effects that cause compilation errors when undesirable compositions are attempted. For example, one approach [36] intentionally excludes certain declarations from undesirable specializations of C++ templates. If these undesirable specializations are attempted, then unresolved reference errors will be reported by the compiler. Similarly, other approaches [36,101] use access control violations in C++ to generate compilation errors if undesirable compositions are attempted.

These ad-hoc approaches only strengthen the case for supporting explicit compositional constraints at the language level. First, ad-hoc approaches have limited expressiveness. For example, constraining subclasses is often difficult when these ad-hoc approaches are used [101]. In addition, defining a specialization in advance to explicitly invalidate each undesirable composition is not practical in large systems. Second, and most importantly, ad-hoc approaches cannot provide informative error messages when constraints are violated. The compiler can only describe the side-effect that aborted compilation, not the semantic condition that was violated. In addition, the error messages produced will vary from compiler to compiler. For usability reasons alone, ad-hoc approaches are inadequate for large-scale programming.

In contrast to ad-hoc approaches, semantic checking is explicitly supported in some GenVoca (§2.3.1) implementations using *design rule*s [13], which model type equation parameterization. In GenVoca, a type equation is represented by a tree whose nodes are the types that appear in the type equation. The tree's edges connect actual type parameters to the node that uses them. All constraint processing is based on the *upflow* and *downflow* of semantic information through this tree. These flows contain attribute name/value pairs that move in both directions in the tree—from leaves to root, and from root to leaves. When necessary, upflows are merged and downflows are split.

GenVoca attribute names are associated with one of four values: *assert*, *negate*, *any* or *inherit*. These values can be modified as they flow through the tree. Predicates, which are defined over attributes, also take one of four values:

true, *false*, *assert* and *negate*. Predicates are combined using conjunction or disjunction to create compound logical expressions. These expressions test whether a semantic constraint holds at a particular node in the tree.

Semantic checking in JL differs from semantic checking in GenVoca implementations in two fundamental ways. First, JL models class hierarchies rather than type equation parameterization, which means that JL can use lists rather than trees to represent its semantic attribute space. This simpler organization allows JL to avoid the splitting and merging of attribute flows that occur in GenVoca. Second, JL's static approach uses attributes that do not have values and ordered attribute lists that are immutable once they are constructed. Constraint checking in JL takes place only after all lists are built. On the other hand, GenVoca's dynamic approach incorporates the notion of data flow and the use of attributes whose values change. Despite its relative simplicity, JL's semantic checking can express all the constraints that have been described in the GenVoca literature to date.

Perry's Inscape [92] environment allows pre-conditions, post-conditions and obligations to be expressed for functions in languages such as C [63]. Inscape's semantic rules can statically guarantee certain properties of a program's runtime state and, to some degree, its correctness. In large systems that have many functions, Inscape may process thousands of rules during compilation.

In contrast with Inscape, JL's semantic checking is designed only to restrict inheritance and the binding of type parameters, both of which are static operations. In JL, pre-conditions, post-conditions and obligations are expressed for

class compositions. These compositions often contain fewer than ten classes and are unlikely to contain more than a few dozen classes, so the number of rules processed at one time is usually limited. JL's goals are less demanding than those of Inscape, so JL's design can be simpler than Inscape's.

CHAPTER 5 CLASS HIERARCHY OPTIMIZATION

In §2.4.4, we described the runtime costs of excessive design-time layering. In this chapter, we address these concerns about mixin performance by presenting the high-level design for the *class hierarchy optimization*. This optimization minimizes the effects of layering in executable code by reducing the number of classes in a hierarchy and by inlining methods. The optimization is designed to work on any valid Java bytecode, not just bytecode generated by Java Layers. In particular, existing Java applications can be optimized.

Our optimization reduces the depth of an existing class hierarchy by recursively combining adjacent classes in the hierarchy. This pairwise *class merging* (or *flattening*, *compressing* or *collapsing*) eliminates the parent class from the hierarchy but preserves the *signature* of the child class. We defer the precise definition of class signature and merging until §5.1, but the basic idea is that the optimized child class can be used in place of the unoptimized child and parent classes with few restrictions.

Figure 45 shows an unoptimized hierarchy consisting of three classes, `A`, `B` and `C` (`Object`, the parent of `A`, is not shown). `A` is the *root* and `C` is the *leaf* of the hierarchy being optimized. After optimization, only the leaf `C` remains.

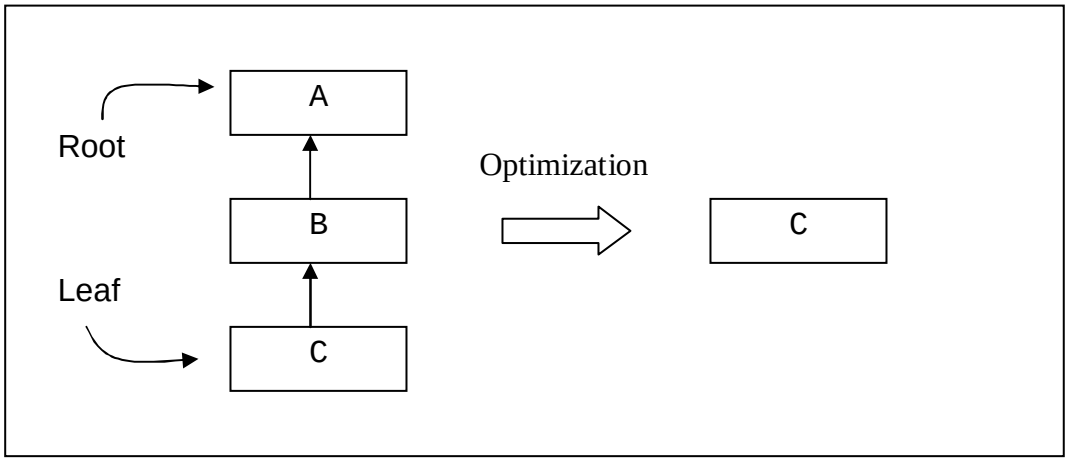


Figure 45 - Simple Class Flattening

Although the idea of merging classes in a hierarchy is simple and intuitive, a number of complicating factors make this an interesting problem. These complicating factors include handling name collisions, adjusting references, guaranteeing accessibility when code is moved across packages, and flattening classes that contain nested classes. The policies used to handle these complications affect the semantics of the optimization, and often more than one policy can be reasonably applied in a situation. In addition to choosing among competing policies, we must also define when the optimization should not be applied and when the simultaneous optimization of multiple hierarchies is necessary.

To illustrate this last point, Figure 46 shows the optimization of parent class `Y` and child class `X`, both of which contain nested classes. In the example, the four hierarchies with leaf classes `X`, `A`, `B`, and `C` are transformed. We merge `Y` into `X`, `Y.A` into `X.A`, and `B'` into `B`. We also relocate `C` into `X`. In general, our optimization will recursively optimize hierarchies at arbitrary nesting depths.

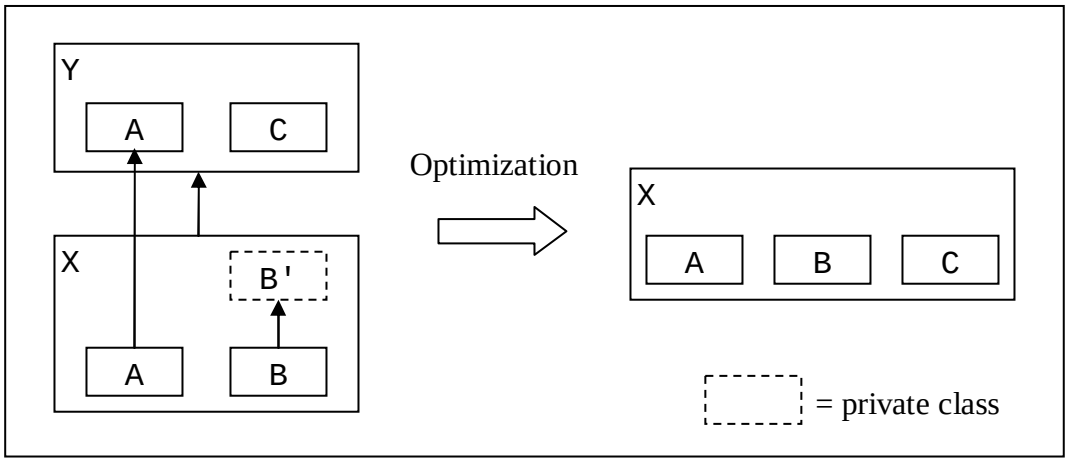


Figure 46 - Nested Class Flattening

The class hierarchy optimization is actually a mapping of unoptimized Java packages to optimized Java packages. The optimization doesn't change or delete existing bytecode; instead all class files that participate in the optimization are copied into a new jar file (Java archive file [8]). This new jar file contains the bytecode of the optimized class hierarchies. The jar file also contains other code from packages that participated in the optimization, the details of which we will describe (§5.1.9).

Our contribution is the high-level design of the class hierarchy optimization. We present a high-level specification that describes the goals, policies and algorithms for implementing the class hierarchy optimization. We do not discuss bytecode representation, how bytecode is manipulated, or how this high-level design is translated into a lower, bytecode-level design. Instead, we rely on the well-defined properties of Java bytecode (§5.2.3) to avoid discussing lower level mechanism.

In addition, the implementation of the class hierarchy optimization is not part of the Java Layers compiler described in this dissertation, nor is the implementation a contribution of this dissertation. The implementation is part of ongoing work related to Java Layers.

Though we can imagine an optimization analogous to our class hierarchy optimization for interfaces, we choose not to investigate interface hierarchy optimization for three reasons. First, we believe that interface hierarchy optimization will have only a marginal effect on application performance. Applications tend to use relatively small numbers of interfaces. Moreover, interfaces contain almost no executable code, which keeps them small and limits opportunities for further optimization. Second, interfaces are often explicitly defined to convey semantic information or to enable a class to be used in certain contexts. Both of these uses reduce the applicability of any interface hierarchy optimization. Finally, multiple inheritance complicates the task of merging interfaces and increases the implementation effort.

We begin our design presentation by defining terminology that we use throughout the discussion. We then formalize our assumptions and list conditions that disable the optimization. Next, we present the general class merging algorithm. We then discuss the details of the algorithm concerned with detecting disabling conditions, updating references, and adjusting access control. We conclude by discussing design alternatives and related work.

5.1 Terminology

In this section, we define the terms and concepts that underlie our design. The policy and algorithm discussions in later sections depend on the precision of the terms defined here. The terminology in this section relies on the Java Language Specification [49], which we abbreviate as JLS in citations.

5.1.1 JAVA TYPES

Non-primitive Java types can be *class* types (JLS §8) or *interface* types (JLS §9). Types that are declared in the body of other types are called *nested types*. *Top-level* types are types that are not nested. A nested type is lexically contained in one or more *enclosing types*; a nested type is declared in the body of its *immediately enclosing type*. Nested classes can be static or non-static; non-static nested classes are also called *inner classes* (JLS §8.1.2).

Paraphrasing JLS §8.1.3, the optional *extends* clause in a class definition specifies the *direct superclass* of the class. A class is said to be a *direct subclass* of the class it extends. If class definition does not have an *extends* clause, then the direct superclass of the class is implicitly *Object* (*Object* is the only class with no direct superclass). The *subclass* relationship is the transitive closure of the direct subclass relationship. Class *P* is a *superclass* of class *C* if *C* is a subclass of *P*.

Superinterfaces and *subinterfaces* are defined in a similar manner for classes (JLS §8.1.4) and interfaces (JLS §9.1.2). We use *supertype* and *subtype* when speaking generically.

When we speak of optimizing class hierarchies in this discussion, we are referring to hierarchies of Java class types since our design does not include interface hierarchy optimization.

5.1.2 INPUT CLASS

The *input class* to the class hierarchy optimization is a top-level class in a class hierarchy that the user specifies. For example, X is the input class for the optimization shown in Figure 46 on page 127.

5.1.3 CLASS MERGING

The *class merge operation* moves the bytecode from superclass *P* into its direct subclass *C* and then eliminates *P*. After the operation, *P* is said to be *merged* into *C*.

During optimization, *n* merge operations and *m* move (without merging) operations, $0 \le n,m$, relocate the bytecode originally in *P*. When the optimization completes, *P*'s bytecode resides in some class *D*. *D* is the *ultimate destination class* for *P*.

5.1.4 PACKAGE RESIDENCE

Package residence extends the concept package membership (JLS §7.1), which includes only top-level types. The *residents* of a package are all types declared in the compilation units (JLS §7.3) of the package, including all nested classes, nested interfaces, local classes and anonymous classes. A type is a package resident if it is a package member or if it is lexically enclosed in a package member.

5.1.5 CHANGE BARRIER

The *change barrier* is the set of packages whose resident types cannot be modified or eliminated during optimization. In hierarchies being optimized, a class is said to be *behind the change barrier* if (1) the class is a change barrier package resident, or (2) the class is a superclass of a change barrier package resident.

The idea of a barrier becomes apparent when traversing a class hierarchy from leaf to root: the first class encountered that is a change barrier package resident immediately causes all its superclasses to be behind the change barrier. Classes behind the change barrier never have their code relocated or inlined by the class hierarchy optimization.

The change barrier always contains all standard packages, including all `java` and `javax` packages [8]. In addition, implementations should allow the change barrier to be augmented with user-specified packages. Since types behind the change barrier never change, packages in the change barrier are not written to the output jar file.

5.1.6 CLASS SIGNATURE

The *signature* of a class consists of (1) the class's fully qualified name (JLS §6.2), (2) the non-private members and non-private constructors that are declared in the class or inherited by the class, and (3) the interfaces implemented by the class or any of its superclasses.

Figure 47 shows parent class `P`, its child class `C`, and the signature of `C`. `P`, `C` and the interfaces they implement are defined in the default package. `C`'s signa-

ture consists of C's name, the members C inherits from P, the non-private members and constructors C declares, and the interfaces implemented by P and C. C does not inherit (JLS § 8.4.6) constructors, private members, or hidden members from P, so they are not part of C's signature.

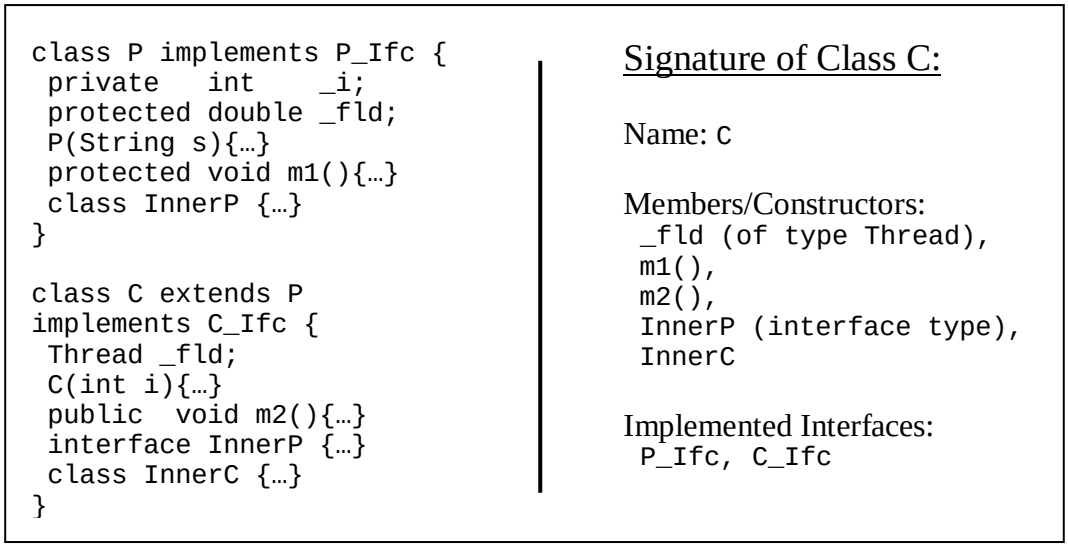


Figure 47 - Signature of Class C

5.1.7 CLASS SIGNATURE PRESERVATION

If class *C* with signature *S* is transformed into class *C'* with signature *S'*, then signature S is *preserved* if (1) *C* and *C'* have the same fully qualified name, (2) all members and constructors in *S* are also in *S'* and (3) all interfaces implemented in *S* are also implemented in *S'*. *S' widens S* if *S'* is signature preserving and *S'* contains constructors, members or implemented interfaces not found in *S*.

The most basic goal of our optimization is to preserve the signature of the input class while collapsing its hierarchy as much as possible. By preserving the

input class's signature, code that uses the unoptimized version of the class can run using the optimized version without being recompiled.

5.1.8 ASSOCIATED NESTED HIERARCHIES

For class C with signature S, let N be the set of nested classes in S. The set of the *associated nested hierarchies* of C is defined as the union of (1) the hierarchies of leaf classes $n \in N$ and (2) the associated nested hierarchies for each class n.

This recursive definition is useful because it represents the set of all hierarchies that can be optimized for a given input class. Specifically, in addition to its own hierarchy, the optimization of input class C recursively optimizes C's non-private nested classes, whether they are declared or inherited. These nested classes are leaf classes in C's associated nested hierarchy set.

Figure 48 shows an example input class `C` and its associated nested hierarchies. All classes in the figure are non-private. All classes in the figure except for `C` itself are part of `C`'s associated nested hierarchies. The leaf classes in `C`'s associated nested hierarchies are `A`, `B`, and `Z`.

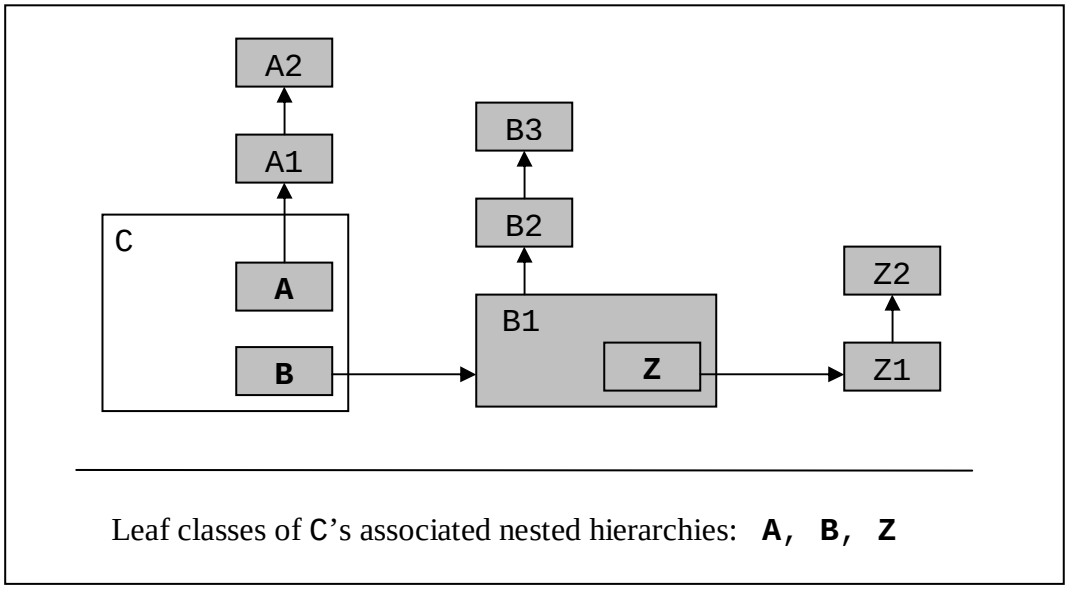


Figure 48 - Class C's Associated Nested Hierarchies

5.1.9 PARTICIPATING PACKAGES

For input class *C*, let *C** be the set of all classes in *C*'s hierarchy or in any of *C*'s associated nested hierarchies. The *participating packages* for input class *C* is the minimal set of packages P^C such that for each $c \in C^*$, *c* resides in some package in P^C. P^C is minimal because each package $p \in P^C$ contains at least one class in *C**.

Less formally, the participating packages for input class *C* are those packages that contain at least one class from a hierarchy being optimized.

5.1.10 PARTICIPATING PACKAGE TYPES

The set of *participating package types* consists of all types that reside (§5.1.4) in participating packages.

Participating package non-hierarchy types are participating package types **excluding** those types in the hierarchies that are being optimized.

Participating package classes are the subset of the participating package types that are classes. Likewise, *participating package interfaces* are the subset of the participating package types that are interfaces.

5.2 Assumptions

In this section, we describe assumptions that we make about the use, input and operating environment of the class hierarchy optimization.

5.2.1 CLOSED WORLD ASSUMPTION

In general, it is not possible to determine where all references to a type reside. Like most executable code, Java bytecode specifies the types on which it depends, not the types that depend on it. Since the class hierarchy optimization changes and even eliminates types, the optimization can invalidate existing references. References will not be invalidated if the user guarantees the following *closed world assumption*:

> **All references to types changed or eliminated by the optimization reside in the input class's participating packages.**

The closed world assumption creates a well-defined set of types that need to be examined and, possibly, altered by the optimization. Types within this set

are kept in a consistent state with regard to any changes made by the optimization; types outside this set are assumed not to be affected by the optimization.

During execution, the class hierarchy optimization loads all bytecode in participating packages. The optimization proceeds under the assumption that all references that need to be considered are in the loaded bytecode, including all reflective references. The restriction imposed by this assumption do not severely limit the use of our optimization use because (1) the intent of class flattening is to eliminate superclasses that are not needed by existing code, (2) the user can designate that specific classes be preserved to maintain existing references (§5.3.2), and (3) the user can stipulate that specific uses of reflection can be ignored (§5.3.11).

5.2.2 SOURCE CODE COMPATIBILITY ASSUMPTION

A compiled Java program is *source code compatible* if its bytecode could be recompiled from legal Java source code, whether or not the source code is actually available. Source code compatibility is stronger then binary compatibility because in addition to successfully loading, verifying and linking, the bytecode does not exhibit any of the source-level inconsistencies described in the JLS §13.

We assume source code compatibility in our high-level design to avoid the inconsistencies that can result from separately evolving class files, which are issues best handled at the implementation level.

5.2.3 JVM PROPERTIES

The transformations described in this specification rely on the architecture of the Java Virtual Machine (JVM) and the binaries it executes, called *class files*.

The Java Virtual Machine Specification [71] defines the instructions (*bytecodes*), symbol tables, and other structures that appear in class files. Since the high-level transformations described in this specification must be implemented in code at the class file level, we need to be sure that class files contain all information required for optimization and that they can be appropriately manipulated.

The class hierarchy optimization modifies class files by creating, deleting, renaming and relocating initializers, constructors and members; by deleting classes; by inlining code; and by adjusting references as a result of the above modifications. Java name obfuscators[11] and application packaging tools [121] currently perform these modifications to class files. These transformations are possible because the Java Virtual Machine Specification (JVMS) guarantees the following bytecode properties:

- *Java is strongly typed.* Every variable and every expression has a type that is known at compile time, though certain checks are performed at runtime (JVMS §2.4).
- *Every type, method and field referenced in a class or interface is represented in the class file of that class or interface* (JVMS §4).[12]
- *All references are symbolic.* In a class file, classes or interfaces, and their fields and methods, are referenced using fully qualified names. For fields and methods, these symbolic references include the name of the class or interface type that declares the field or method, as well as

[11] Two of the many available obfuscators are at www.condensity.com and www.preemptive.com.
[12] In the JVMS, initializers and constructors are treated as special kinds of methods.

the name of the field or method itself, together with appropriate type information (JVMS §2.17.3).

- *Class files contain the complete implementations of classes or interfaces, including references to all their declarations* (JVMS §4).

Taken together, the above properties guarantee that our optimization has access to all the information it requires to perform its code transformations. In a class file, references are stored as fully qualified names in the *constant pool* (JVMS §4.4). During class merging, we can locate all references to any type, method or field by searching the names in the constant pools of all participating package types. When we change a declaration's name or location, we can perform the appropriate fix-up at all sites that reference that declaration. During method inlining, we use constant pool information in the same way that current JVMs do when they inline code [9,61].

5.3 Disabling Conditions

This section describes the *disabling conditions* that prevent a superclass from being merged into its direct subclass. Many disabling conditions are needed to maintain existing class references; others are needed to preserve the nested structure of lexically related classes. Classes without disabling conditions are said to be *enabled for optimization*, which allows these classes to be eliminated from the optimized code. We have identified the following eleven disabling conditions.

5.3.1 SIGNATURE PRESERVATION

A class cannot be merged if it is part of the input class's signature.

In Figure 49, input class X contains public nested classes A and B. B cannot be merged into A because B is public and, therefore, part of the signature of X that the optimization is preserving.

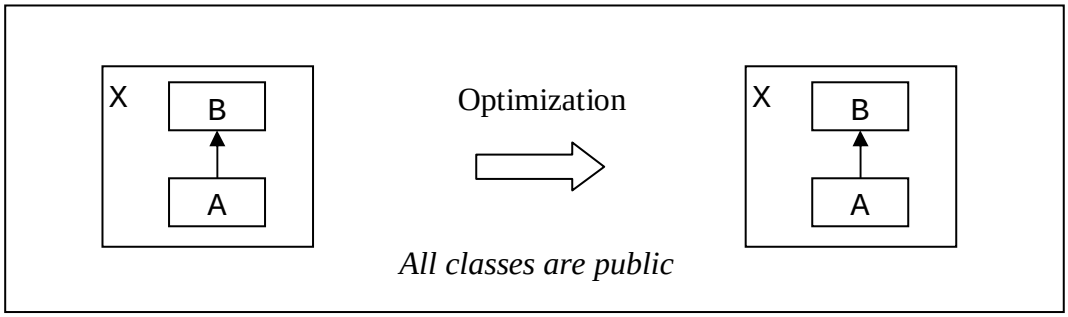


Figure 49 - Signature Preservation Disables Optimization

5.3.2 USER-SPECIFIED PRESERVATION

The user can specify that any class be preserved, and the optimization will not merge that class.

5.3.3 CHANGE BARRIER PRESERVATION

Classes behind the change barrier cannot be merged or changed in any way.

5.3.4 CHANGE BARRIER REFERENCES

Classes referenced from behind the change barrier cannot be merged. This restriction is necessary because references behind the change barrier cannot be adjusted to refer to the merged class.

5.3.5 MERGING STATIC AND NON-STATIC CLASSES

A static superclass cannot be merged into a non-static subclass. Static classes can have static members, which are generally prohibited in non-static classes (JLS §8.1.2).

A non-static superclass cannot be merged into a static subclass. Non-static classes can reference instance members of enclosing classes, something which static classes are prohibited from doing (JLS §8.5.2).

Top-level classes are treated as static when these conditions are tested.

5.3.6 EXPLICIT ALLOCATION

Class types explicitly allocated with the `new` keyword in a participating package type cannot be merged.

The justification for this condition is as follows: When parent class *P* is merged into its child class *C*, *P* is eliminated and all existing references to *P* are replaced with references to *C*. This substitution requires the selection of an appropriate constructor for *C* in allocation expressions. Since constructor selection cannot be performed using only syntactic information, the merging of *P* is not permitted.

5.3.7 MULTIPLE SUBCLASSES

Classes with multiple subclasses in the participating packages cannot be merged.

The justification for this condition is as follows: When parent class *P* is merged into its child class *C*, *P* is eliminated and all existing references to *P* are replaced with references to *C*. If another subclass, *C'*, of *P* exists, then construc-

tors in *C'* would now have to call appropriately selected constructors in *C*. Since constructor selection cannot be performed using only syntactic information, the merging of *P* is not permitted.

5.3.8 LEXICAL NESTING

The Lexical Nesting condition helps preserve the lexical relationship between enclosing and nested classes during optimization. This preservation is important because lexically related classes have access to each others private members, access that must be maintained in optimized code.

We define the Lexical Nesting condition in terms of three classes. Let class *EnclosingP* contain nested member class *P*, and let *C* be a direct subclass of *P*. If the ultimate destination class of *C* is immediately enclosed in the ultimate destination class of *EnclosingP*, then *P* satisfies the Lexical Nesting condition. Otherwise, *P* cannot be merged into *C*, and *P* is said to violate the Lexical Nesting condition.

Figure 50 shows the simplest configurations for merging nested classes. In optimization (a), `P` and `C` are both nested in `X`. The ultimate destination of `C` is `C`, which is enclosed in `X`. The ultimate destination of `P`'s enclosing class (`X`) is also `X`, so the Lexical Nesting condition is satisfied.

In optimization (b), the pattern of inheritance is more complex but still regular: Enclosing classes form one hierarchy and nested classes form another. The enclosing class of the ultimate destination of `C` and the ultimate destination of `Y` are the same class (`X`), so the Lexical Nesting condition is satisfied.

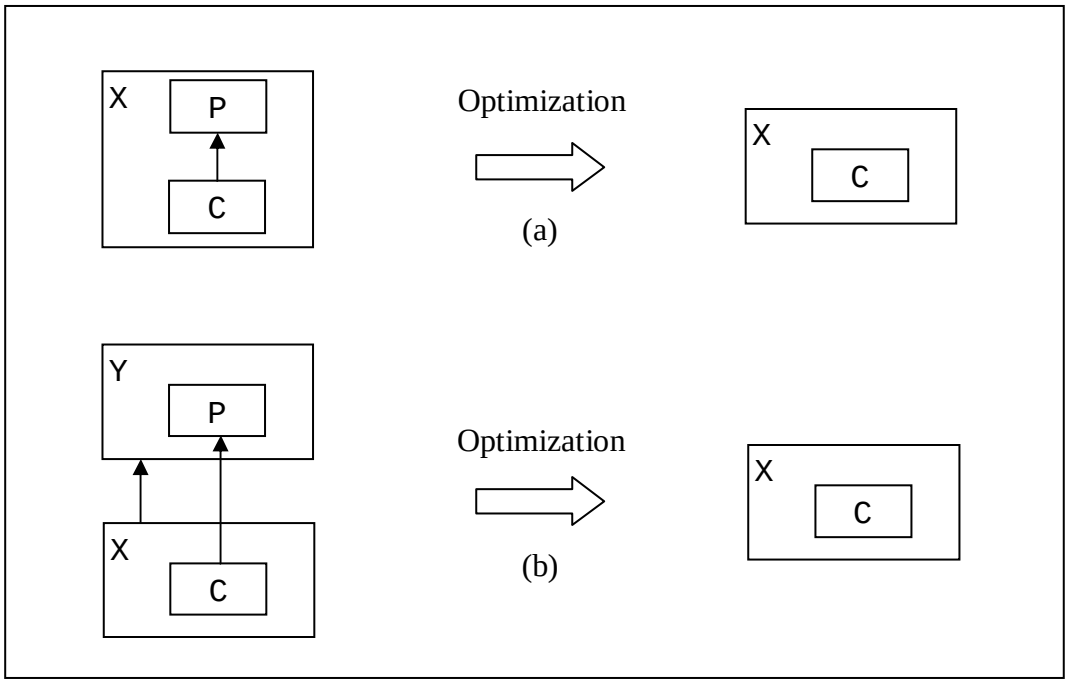


Figure 50 - Simple Nested Class Merging

Figure 51 shows that more complex configurations of nested class also can be merged. Optimization (c) is like optimization (b) in Figure 50, except `J` now intervenes between `X` and `Y` in the enclosing class hierarchy. The enclosing class of the ultimate destination of `C` and the ultimate destination of `Y` (and `J`) are the same class (`X`), so the Lexical Nesting condition is satisfied.

In optimization (d), `K` intervenes between `P` and `C` in the nested class hierarchy. The ultimate destination of both `P` and `K` is `C`. The enclosing class of `C` and the ultimate destination of `Y` are the same class (`X`), so the Lexical Nesting condition is satisfied.

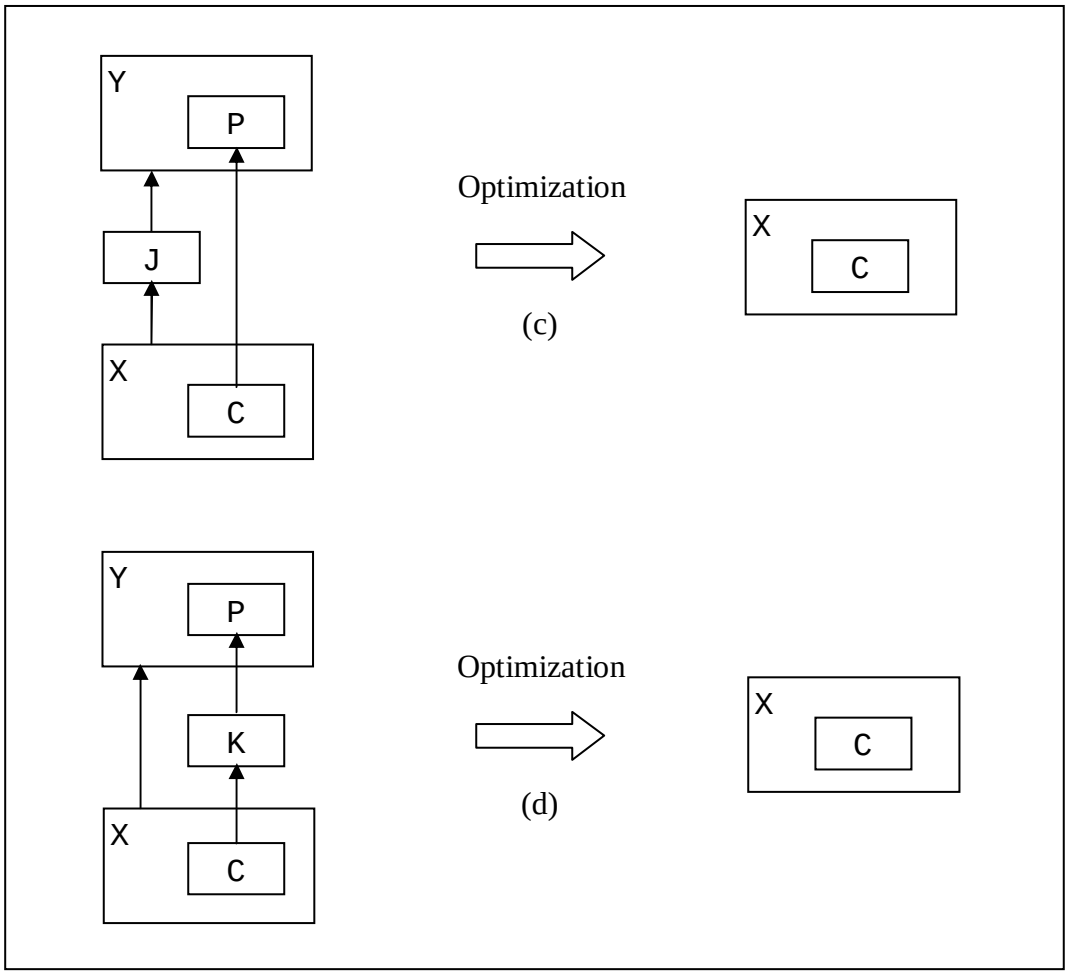


Figure 51 - More Complex Nested Class Merging

Figure 52 shows configurations in which the Lexical Nesting condition disables the merging of class P. In case (e), P is disabled because Y is not merged into X, which distinguishes this case from case (b) above. In case (f), a disabling condition has previously been discovered in J. Since J is disabled, Y is prevented from being merged into X, which disables P.

In case (g), P is disabled because C is not *immediately* enclosed by X. The immediacy requirement in the Lexical Nesting condition maintains the relative nesting positions of enclosing and nested classes, which preserves original code structure during optimization. This preservation simplifies the use and implementation of our optimization because optimized code has a predictable structure that is relatively easy to ensure.

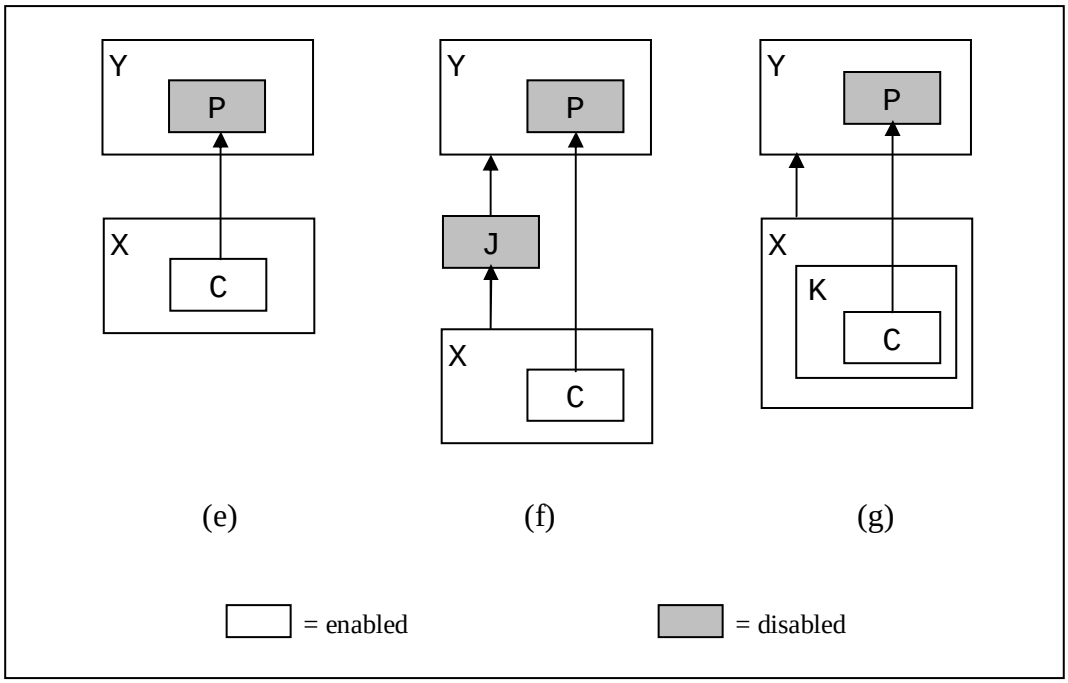


Figure 52 - Disabled Nested Class Configurations

5.3.9 ENCLOSING CLASS

Let class *E* enclose nested type *N*, which can be an interface or class nested at any depth in *E*. The Enclosing Class condition states that *E* cannot be merged if *E*'s ultimate destination class would be enclosed in a subtype of *N*.

An important consequence of the Enclosing Class restriction is that optimized enclosing classes cannot inherit from their nested classes. In Figure 53, on the left, class `E` cannot be merged into its subclass `M`. Here, the ultimate destination class of `E` is `M`, which is enclosed in subclass `D` of `N`. If this condition were not enforced, the configuration on the right of Figure 53 would result from merging `E` into `M`. This configuration cannot be generated by legal Java source code because `D` inherits from its nested class `N`, so we prohibit it in the class hierarchy optimization.

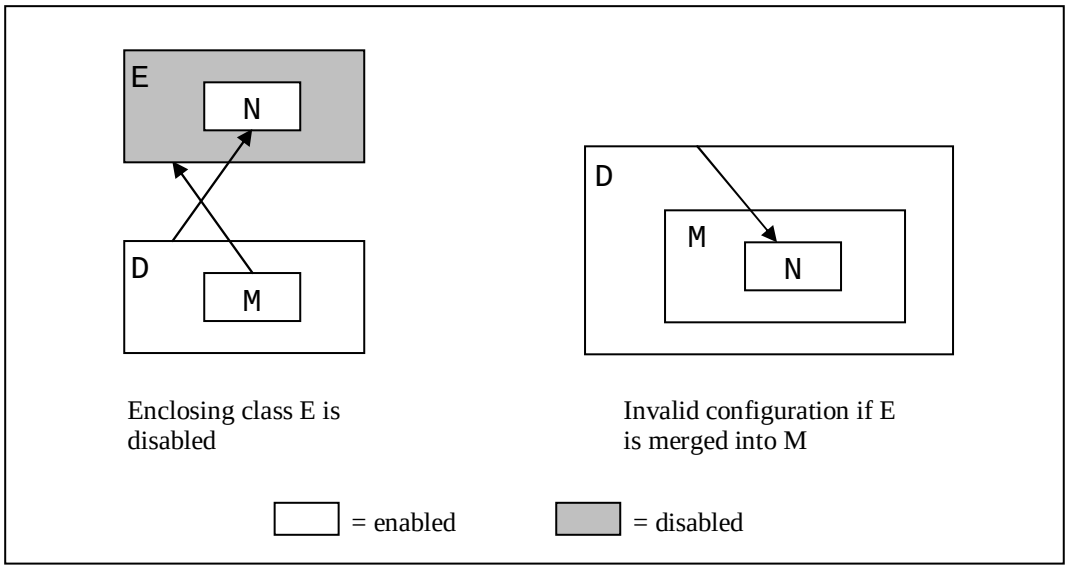


Figure 53 - Enclosing Class Condition

Together, the Lexical Nesting and Enclosing Class conditions preserve the relationship between enclosing and nested classes during class merging. If we ignore the purely syntactic manipulations of references and names, the Lexical Nesting condition guarantees that the bytecode of a nested class is immediately

enclosed in the same bytecode after merging as before merging. Similarly, the Enclosing Class condition guarantees that after merging, an enclosing class never subtypes a type it contains.

5.3.10 CLASS LITERAL USAGE

The use of a class name in a *class literal* (JLS §15.8.2) expression prevents the specified class from being merged. Class literals are expressions of the form `N.class`, where `N` is the name of a class, interface, array or primitive type. These expressions evaluate to the `Class` object of the named type, which requires that the type exists.

5.3.11 REFLECTION

A *reflective call* in a participating package type that has not been designated as safe is a disabling condition for all types being optimized. Reflective calls are calls to `java.lang.Class` or to any class in the `java.lang.reflect` package [8]. We now describe the tradeoffs involved in supporting reflection and the rationale behind our approach.

The challenge of optimizing Java hierarchies in the presence of reflection is to strike a reasonable balance between safety and utility. Since the types referenced by reflective code can depend on input data, determining which types are reflected upon is undecidable in general. Optimizations that modify or eliminate types can change the behavior of reflective code and, therefore, are unsafe. On the other hand, reflective code often inspects types outside the input class's participating packages and, therefore, would not be affected by our optimization.

Our goal is to guarantee safety while at the same time not unnecessarily restricting optimization.

To balance these opposing concerns of safety and utility, we borrow from the approach used by Tip et al. in the Java Application Extractor (Jax) [121]. We abort optimization when reflection is used in any participating package type, excluding types in the standard packages (e.g., `java` and `javax`). Before aborting, the location and description of every reflective call encountered is displayed. The user can then designate reflective calls, individually or in groups, as safe and rerun the optimization. This approach is conservative by default, simple because it avoids trying to understand how reflection is being used, and flexible because reflection can be ignored under the user's direction.

5.4 General Merging Algorithm

In this section, we present the algorithm for merging a superclass into its subclass. We begin our discussion by restating the goals of the class hierarchy optimization using the terminology defined in §5.1. Our goal is to compress a class hierarchy into as few classes as possible while preserving the signature of the input class and the signatures of the leaf classes of all associated nested hierarchies. Method inlining is performed as hierarchies are compressed.

The class hierarchy optimization is useful only if it *safely* transforms code. Safe transformations preserve program behavior and allow optimized classes to be used in place of their unoptimized counterparts. In §5.2, we described the assumptions that our optimization makes and that users must guarantee for safe exe-

cution. In §5.3, we described conditions that our optimization detects that prevent types from being merged unsafely. We now describe our algorithm.

5.4.1 ALGORITHM OVERVIEW

Our class merging algorithm consists of three stages. The initialization stage loads into memory all data needed during optimization. This data is analyzed and organized for later use. Next, the identification stage detects the portions of hierarchies that can be optimized. Identification includes testing classes for disabling conditions. Finally, the merging stage actually collapses superclasses into their subclasses and then eliminates the superclasses. In this stage, class members are moved and renamed, references are updated, methods are inlined, and access control is adjusted. We now present our procedure for class hierarchy optimization, starting with initialization.

5.4.2 INITIALIZATION

The goal of initialization is to load into memory and then organize all data needed during optimization. Let class *C* be the input class to the optimization procedure. The procedure terminates immediately if *C* is behind the change barrier. Otherwise, initialization proceeds as follows:

1. Load *C*'s class file.
2. Load the class files of all of *C*'s superclasses.
3. Load the class files of *C*'s associated nested hierarchies.
4. Construct the participating package set.
5. Load the participating package types.

6. Set each class's *mergeable* attribute.
7. Initialize *analysis information*.

We define *H* to be the set of class hierarchies consisting of input class *C*'s hierarchy and all associated nested hierarchies of *C* (§5.1.8). For convenience, we refer to "classes in the hierarchies of *H* " as "classes in *H*."

We construct the set of participating packages for *C* by aggregating the packages of all classes in *H*, using package residence (§5.1.4) as our selection criterion. Once we know the participating packages, we load all participating package types that have not been previously loaded.[13] Since types in the standard packages like `java` or `javax` cannot contain references to classes in *H*, we do not load resident types from standard packages.

Each participating package type has a *mergeable* attribute that indicates whether the class can be merged into its subclass. The possible values for *mergeable* are *enabled*, *disabled*, or *undecided*. We say a type is enabled, disabled, or undecided depending on the value of its *mergeable* attribute. During initialization, all classes in *H* are undecided. All other participating package types are disabled. Undecided classes will be enabled or disabled before any class merging occurs (§5.4.3).

Lastly, initialization also extracts *analysis information* from participating package types. Analysis information consists of the reflective data and the reference data used during later stages of the optimization. We will introduce the vari-

[13] Loading participating package types is implementation dependent and usually requires the ability to load all class files from file system directories.

various kinds of analysis information as we encounter them. Analysis data can be collected during initialization or at any time before they are actually needed. Once initialization is complete, we determine the classes that can be optimized.

5.4.3 IDENTIFYING OPTIMIZABLE HIERARCHIES

After initialization, we identify the classes that can be merged into their subclasses. All classes in H are candidates for optimization, but some may have disabling conditions that prevent merging. We define a new set H' to contain the *fragments* of the hierarchies in H that can be merged. A fragment of a Java class hierarchy is a subset of the hierarchy that contains a leaf class, a root class, and all intermediate classes on the inheritance path between the leaf and root. Fragments are the basic unit of optimization for the algorithm described in §5.4.4, which merges a fragment's non-leaf classes into its leaf class.

We begin our discussion by defining the *outside-in processing order*, which is used in constructing H'. We then present the algorithm that builds H'.

5.4.3.1 Outside-In Processing Order

A key characteristic of the H' construction algorithm is that it implements the *outside-in processing order*. This ordering specifies that (1) if L is a leaf class in H and class E in H encloses L, then E is processed before L, and (2) the next class processed after class A is A's direct superclass or, if no superclass exists, an unprocessed leaf class in H. The ordering gets its name from the fact that in H, enclosing (outside) classes are processed before their nested (inside) leaf classes.

The outside-in processing order is useful in detecting disabling conditions, which we discuss in §5.5. In particular, the outside-in processing order is used as

a heuristic that increases the accuracy of the *leaf()* method, which we define in §5.5.4.1. The *leaf()* method is used to detect the Lexical Nesting (§5.5.4) and Enclosing Class (§5.5.5) disabling conditions.

5.4.3.2 The *H'* Construction Algorithm

The pseudo-code below specifies how *H'* is constructed. The algorithm inspects the input class's hierarchy first and then iteratively inspects nested class hierarchies. Classes found to have disabling conditions become leaf classes in fragments in *H'*. We now present the *H'* construction algorithm.

```
 1. H' = H.copy()
 2. c  = input class
 3. worklist = an empty list
 4. Fragment(c, worklist)
 5. while worklist not empty
 6.    c = worklist.pop()
 7.    Fragment(c, worklist)
 8. terminate construction algorithm

 9. procedure:  Fragment(c, worklist)
10. while c != null and c.isUndecided()
11.    if c has a disabling condition
12.       then c.disable()
13.       else c.enable()
14.    if c.isDisabled() and c is not a leaf class in H'
15.       then make c the leaf class of a new fragment in H'
16.    append to worklist the nested classes in c that are leaf classes in H
17.    c = c's superclass or null if c has no superclass
```

After initialization (§5.4.2), all classes in *H* are undecided. The above algorithm begins by populating *H'* with a copy of the contents of *H*, assigning the input class to *c*, and creating the empty *worklist* (lines 1-3). Lines 4-8 drive the

algorithm and implement the outside-in processing order. Lines 9-17 define the *Fragment* procedure that creates fragments in *H'*.

In line 4, the *Fragment* procedure is called with the input class and the empty *worklist* as parameters. Upon return, if *worklist* contains nested classes, the while loop on lines 5-7 executes. Line 6 removes the first class from *worklist* and assigns it to *c*. On line 7, *Fragment* is called again. Any call to *Fragment* causes the number of classes in *worklist* to either increase or stay the same. Eventually, the loop in lines 5-7 terminates because all classes in (finite) *H'* are processed. Line 8 terminates with *H'* completely constructed.

The *Fragment* procedure on line 9 consists of a while loop (lines 10-17) that processes all classes in *H'* exactly once. The *mergeable* attribute for class *c* is changed in lines 11-13 from *undecided* to either *enabled* or *disabled*, which prevents *c* from being processed more than once. Lines 14-15 create a new fragment with disabled leaf class *c* if *c* is not already a leaf. Line 16 appends *c*'s nested classes to *worklist*, but only those that were leaf classes in *H* (i.e., only associated nested hierarchy leaf classes). Line 17 controls the traversal of *c*'s hierarchy.

A key characteristic of the construction algorithm is that it implements the outside-in processing order (§5.4.3.1). The *Fragment* procedure imposes this ordering in two ways. First, the procedure processes a class's complete hierarchy before returning, which satisfies the second condition of the ordering definition. Second, the *Fragment* procedure populates *worklist* with the nested leaf classes it encounters as it traverses a class's hierarchy. Since nested leaf classes only appear in *worklist* if their enclosing class has been processed, the first condition of

the outside-in ordering definition is also satisfied. The construction algorithm begins by processing the input class, which is the top-level class that spawned *H*, and then processes the *worklist*.

Figure 54 illustrates the effect of the construction algorithm. Fragments in *H'* represent the groups of classes from the original hierarchies in *H* that can be merged with each other.

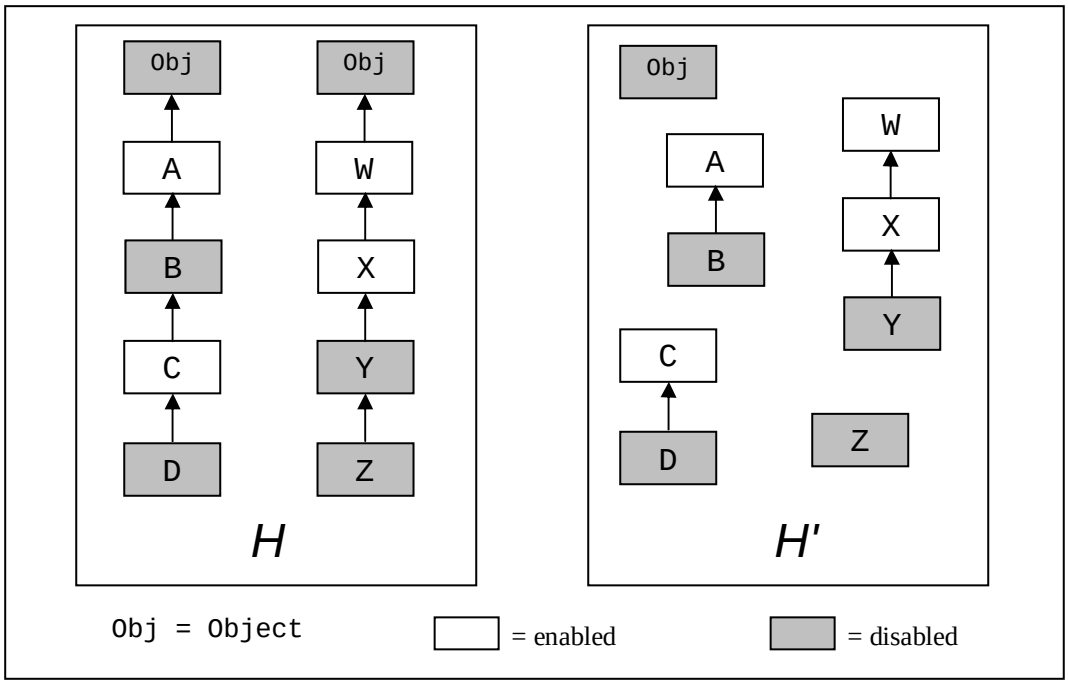


Figure 54 - Identifying Optimizable Fragments

H' encodes important information for detecting disabling conditions (§5.5) and for bytecode merging (§5.4.4). First, all class merging takes place in the context of some fragment in *H'* or, conversely, each fragment in *H'* represents zero or more merge operations. Second, all classes in fragment $h' \in H'$ are merged into the leaf class of *h'*. This means that a class *c* is an ultimate destination class only if *c*

is the leaf class of some fragment in *H'*. Lastly, fragments can be processed in any order because all required reference information (i.e., the set of ultimate destination classes) is known as soon as *H'* is constructed.

We defer our description of disabling condition detection until §5.5, and continue by discussing how fragments are merged once *H'* is constructed.

5.4.4 CLASS MERGING

Classes in the fragments in *H'* are merged in a pairwise manner, starting at the leaf, according to the following procedure:

```
1. for each h'∈ H'
2.   leaf = leaf class of h'
3.   sc = superclass in h' of leaf or null if none exists
4.   while sc != null
5.     MergePair(leaf, sc)
6.     sc = superclass in h' of leaf or null if none exists
```

Merging only takes place between classes in the same fragment. In the above procedure, each fragment *h'* in *H'* is collapsed into its leaf class. The class hierarchy optimization completes when all fragments *h'* in *H'* have been merged.

Step 5 above merges the immediate superclass of a leaf class into the leaf class. This pairwise merge is repeated until all superclasses in the fragment have been consumed. The bulk of the work in merging classes occurs in this step, which we now describe in detail.

5.4.5 PAIRWISE CLASS MERGING

In this section, we describe the pairwise merging algorithm in which superclass *P* is merged into its direct subclass *C* in fragment *h'*∈ *H'*. The algorithm

in §5.4.4 guarantees that *C* is always the leaf class in *h'*. The pairwise merging algorithm has two steps.

5.4.5.1 Step 1: Move Code

We move bytecode definitions from *P* to *C*. These definitions include all initializers, constructors, fields, methods and interfaces. In addition, we also move some of *P*'s nested classes into *C*. Specifically, all nested classes in *P* are moved into *C*, except those that are enabled for optimization. We can defer processing enabled nested classes until the main merging algorithm selects their fragment (§5.4.4). This deferral avoids moving code that will later be merged, which increases the algorithm's efficiency.

Moving bytecode definitions between classes often requires renaming or other transformations. For example, members in *P* that were previously hidden by members in *C* are renamed upon relocation to avoid name conflicts. Constructors in *P* are converted to private methods in *C* to preserve *C*'s signature. In addition, references to relocated definitions are updated to reflect the definitions' new locations. We defer the discussion of these transformations until §5.6.

Another type of transformation caused by code movement has to do with access control. When *P* and *C* are in different packages, access control often needs to be adjusted to preserve the validity of references. These adjustments are performed on relocated definitions, on *C* itself, and on constructors and members in participating package types. The details of access control adjustment are described in §5.7.

5.4.5.2 Step 2: Update Supertypes

We replace *C*'s superclass with the superclass of *P*. We also selectively add the interfaces implemented by *P* to the list of interfaces implemented by *C*. Each interface *I* implemented by *P* is added to *C*'s implementation list if *C* does not already implement *I* or some sub-interface of *I*.

Our discussion of the general merging algorithm is complete, though a number of details still need to be explored. The next section describes how disabling conditions are detected during the construction of *H'*. Subsequent sections describe how code is transformed and how access control is adjusted during class merging.

5.5 Detecting Disabling Conditions

In this section, we describe how disabling conditions (§5.3) are detected in classes in *H'* (§5.4.3). The analysis required to detect disabling conditions differs from conventional compiler analysis [2,7,81] because it is not concerned with data flow, control flow, or resource utilization. Instead, our analysis is concerned with the structure of class hierarchies and its goal is to determine if a hierarchy can be transformed and if classes can be eliminated.

Our goal is to show that we can test for all the disabling conditions listed in §5.3 as we construct *H'* using the algorithm defined in §5.4.3. Specifically, the tests we describe all take place in the context of line 11 of the construction algorithm on page 151. We assume that all set up and initialization for the construction algorithm is complete. In particular, we only test for disabling conditions on

classes whose *mergeable* attribute is undecided (§5.4.2). After our tests, the *mergeable* attributes in all participating package types are set to either enabled or disabled.

To be enabled for optimization, an undecided class must not exhibit any disabling condition. Disabling conditions can be tested in any order because they are independent of each other. As soon as a violation of any condition is detected, the class is disabled and no further testing for that class is performed.

Tests for some disabling conditions require the extraction of *analysis information* (§5.4.2) from the class files of the participating package types. We define the analysis information needed for a test as we describe that test's implementation.

5.5.1 DETECTION USING NO ANALYSIS INFORMATION

Testing for Signature Preservation (§5.3.1) simply means disabling all original leaf classes in H'. Testing for User-Specified Preservation (§5.3.2) means disabling any classes the user explicitly told us to preserve. Testing for Change Barrier Preservation (§5.3.3) means that as we traverse the hierarchies in H', we disable classes that are behind the change barrier (§5.1.5).

5.5.2 DETECTION USING REFLECTIVE INFORMATION

Reflective information is the subset of analysis information that can be retrieved using Java's reflective APIs. Before testing for disabling conditions, we count the number of subclasses each class in H' has in the set of participating package types. Testing for Multiple Subclasses (§5.3.7) means disabling any class in H' that has more than one subclass. Testing for Merging Static and Non-

Static Classes (§5.3.5) means that the modifiers on each class in H' are inspected during traversal. A superclass is disabled if its subclass has an incompatible static designation.

5.5.3 DETECTION USING REFERENCE INFORMATION

In this section, the disabling condition tests rely on analysis information extracted by scanning the class files of the participating package types. Implementations will most likely scan class files once during initialization and collect all the data that will be needed later. For this discussion, let P be the class we are testing for disabling conditions.

Testing for Change Barrier References (§5.3.4) means disabling P if any class behind the change barrier contains a reference to P. Testing for Explicit Allocation (§5.3.6) means scanning all loaded participating package types for the use of P in allocation statements and disabling P if a use is found. Testing for Class Literal Usage (§5.3.10) means scanning all loaded participating package types for references to P's class file and disabling P if a use is found.

Lastly, testing for Reflection (§5.3.11) means scanning all loaded participating package types for reflective calls. Each call found is checked against a user-supplied list of reflective calls that should be ignored. If a reflective call is found that is not explicitly ignored, the optimization aborts and no classes are merged.

5.5.4 DETECTING THE LEXICAL NESTING CONDITION

This section describes the Lexical Nesting (§5.3.8) test, which is more complex than previous tests because two hierarchies in H' are considered at once.

We begin our discussion by defining the *leaf()* method, which we use in the algorithm that detects Lexical Nesting. After defining this algorithm, we describe how it depends on the outside-in processing order of the *H'* construction algorithm (§5.4.3).

5.5.4.1 The *leaf()* Method

The *leaf()* method is defined on all participating package types as follows: If type *T* is enabled, then *T.leaf()* returns the leaf class of the fragment in *H'* in which *T* resides. If *T* is disabled, then *T.leaf()* returns *T*. Otherwise, *T.leaf()* returns **null**.

The *leaf()* method allows us to approximate the value of ultimate destination classes (§5.1.3) while we build *H'*. In §5.5.4.3, we describe how the *leaf()* method uses the outside-in processing order.

5.5.4.2 The Lexical Nesting Test

We now describe the test that detects the Lexical Nesting condition. Let class *EnclosingP* contain nested class *P*, and let *C* be a direct subclass of *P*. We apply the Lexical Nesting test shown below to *P* to determine if *P* can be merged into *C*. The test terminates as soon as *P* either satisfies the condition or is disabled.

1. If *EnclosingP* is undecided, we disable *P*.
2. If *C.leaf()* has an immediately enclosing class, we assign this enclosing class to *EnclosingCLeaf*. Otherwise, we disable *P*.
3. If *EnclosingP.leaf()* equals *EnclosingCLeaf.leaf()*, *P* satifies the Lexical Nesting condition. Otherwise, we disable *P*.

In Step 1 of the test, an undecided *EnclosingP* indicates that *P* is being processed before its enclosing class has been processed. In this case, the ultimate destination class for *EnclosingP* cannot be accurately characterized, so *P* is conservatively disabled.

In Step 2, *C*'s *leaf()* method is called. If *C* is enabled, then *C.leaf()* returns the leaf class of the fragment in *H'* in which *C* resides. If *C* is disabled, then *C.leaf()* returns *C*. In §5.5.4.3, we explain why *C* is guaranteed to be enabled or disabled at this point in the processing, which means *C.leaf()* always returns a class. If the class returned is nested, we assign its enclosing class to *EnclosingCLeaf*. Otherwise, the Lexical Nesting condition is violated and *P* is disabled.

In Step 3, the *leaf()* method is used twice in a comparison. Step 1 guarantees that *EnclosingP.leaf()* returns a class. If this returned class matches the value returned from *EnclosingCLeaf.leaf()*, then the two enclosing classes have the same leaf class in *H'*. By having the same leaf class, the enclosing classes also have the same ultimate destination class.[14] In this case, the Lexical Nesting con-

[14] If a leaf class is a top-level class, then it is also an ultimate destination class. If a leaf class is nested, however, it can still be *moved* (but not merged) into other enclosing classes by the optimization. Nested leaf classes in *H'* become ultimate destination classes after all moves are performed.

dition is satisfied. Otherwise, if the enclosing class values do not match, *P* is disabled.

5.5.4.3 Using the Outside-In Processing Order

The Lexical Nesting test relies on the outside-in processing order (§5.4.3.1) implemented by the *H′* construction algorithm (§5.4.3.2). Recall that the outside-in processing order specifies that (1) if *L* is a leaf class in *H* and class *E* in *H* encloses *L*, then *E* is processed before *L*, and (2) the next class processed after class *A* is *A*'s direct superclass or, if no superclass exists, an unprocessed leaf class in *H*.

The outside-in processing order is used in two ways. First, we note that classes already processed by the *H′* construction algorithm are either enabled or disabled. If class *c* has been processed, then (1) *c* resides in some fragment $h' \in H'$, and (2) all of *c*'s subclasses in *h′* have also been processed. In particular, the leaf class of *h′* has been processed and is known. This allows us to define *c.leaf()* to return the leaf class of *h′*. Above, in Step 2 of the Lexical Nesting test in §5.5.4.2, *C.leaf()* returns the appropriate leaf class in *H′* because *C* has been processed.

Second, the outside-in processing order is used as a heuristic to increase the likelihood that *EnclosingP* has already been processed, which means we can often avoid the conservative assignment in Step 1 of the test. For example, Figure 55 shows a typical configuration of mixin layers. When *H′* is constructed, the enclosing class hierarchy with leaf X is processed before either nested class hierar-

chy, which have leaves A and D. As a result, when the nested classes are processed, the Lexical Nesting condition can be precisely tested (Step 2 above).

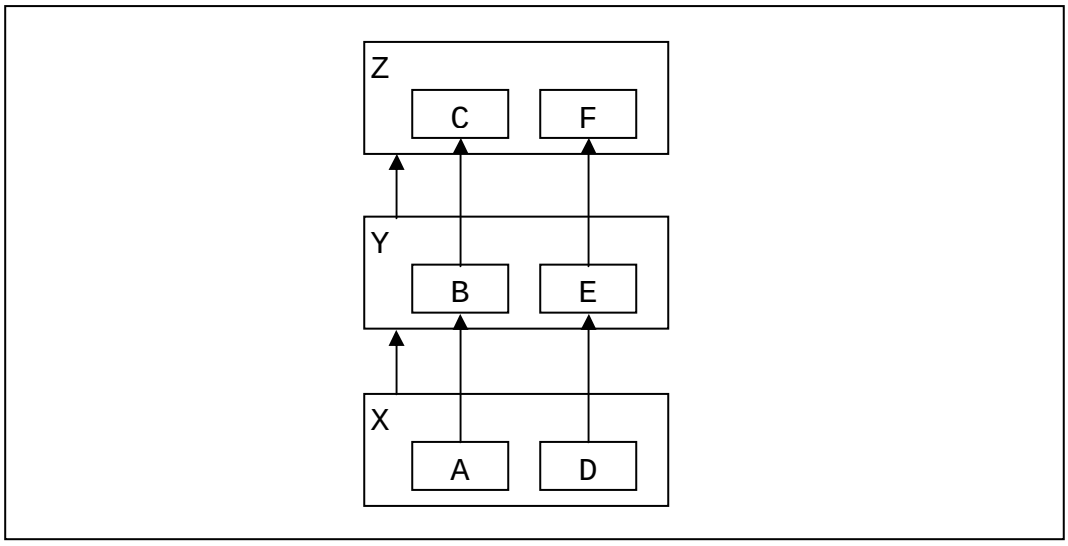


Figure 55 - Application of Outside-In Processing Order

Figure 56, on the other hand, illustrates a configuration in which a nested class is processed before its enclosing class. The hierarchy with leaf U is processed first by our ordering, which means that the Lexical Nesting tests on classes K and L encounter an undecided enclosing classes V and W. In this situation, we conservatively disable merging on K and L in Step 1 in the Lexical Nesting test.

As an alternative, we could process leaf class J's hierarchy first to avoid encountering undecided enclosing classes.[15] We do not claim, however, that undecided enclosing classes can always be avoided.

A precise mathematical treatment of the outside-in processing order, as well as other orderings and approaches to testing the Lexical Nesting condition,

[15] Under this new processing order, L is enabled for merging instead of disabled.

are subjects for further research and beyond the scope of this dissertation. We simply note that the outside-in processing order is simple and works well in regularly structured hierarchies like those in Figure 55, which are the cases that we have encountered in practice using JL.

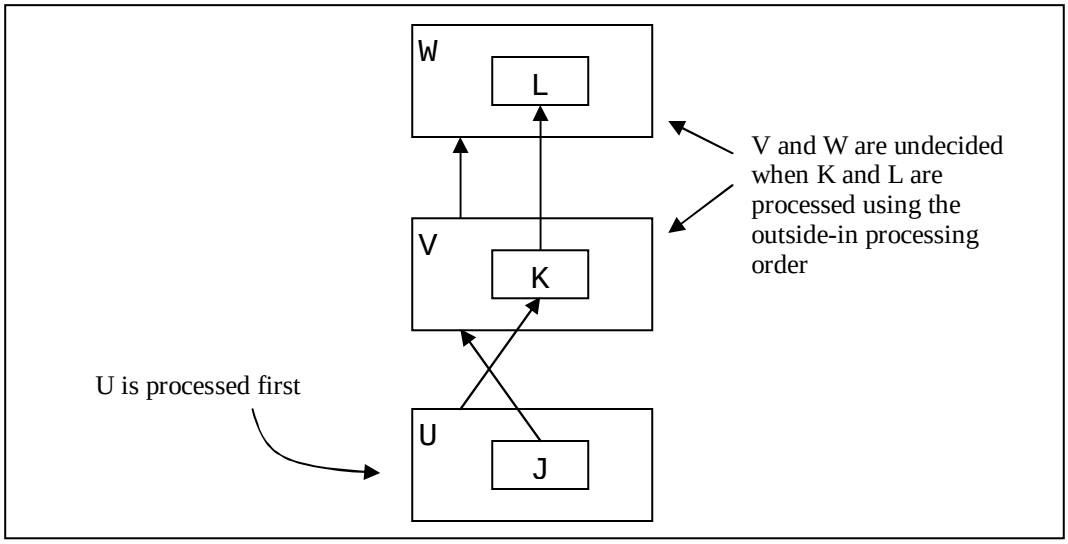


Figure 56 - Order-Defeating Configuration

5.5.5 DETECTING THE ENCLOSING CLASS CONDITION

The Enclosing Class condition (§5.3.9) is tested whenever classes that contain at least one nested type are processed. We begin our test by letting *P* be a class that contains a nested type. The Enclosing Class condition states that *P* can be merged into its direct subclass *C* only if *C*'s ultimate destination class is not enclosed in a subtype of a type enclosed in *P*.

To implement our Enclosing Class test, we use the *leaf()* method defined in the Lexical Nesting discussion in §5.5.4.1. As in that discussion, we rely on

the outside-in processing order of the *H'* construction algorithm (§5.4.3). This ordering guarantees that *C.leaf()* returns a (non-**null**) class. Please refer to §5.5.4 for details.

We let leaf class *L* = *C.leaf()*. If *L* is a top-level class, then *P* satisfies the Enclosing Class condition and the test terminates. If *L* is a nested class, we create the set *NestedTypes*, which contains the transitive closure of all types enclosed by *P*. That is, *NestedTypes* contains all types nested in *P* at all nesting depths. We then apply the following procedure to each enclosing type *E* of *L*, starting with *L*'s immediate enclosing type and working outward. The test terminates when all enclosing types satisfy the procedure below or when *P* is disabled.

1. If *E* is undecided, then we disable *P*.
2. If *E.leaf()* is a subtype of a type in *NestedTypes*, then we disable *P*. Otherwise, *E* satisfies this procedure.

One subtlety in the above algorithm is that we use *leaf()* to discover *P*'s ultimate destination class. We can use *leaf()* in this way because as long as enclosing class *E* is enabled or disabled, *E.leaf()* returns its ultimate destination class *at that level of nesting*. By applying *leaf()* at each successive nesting level starting at *L*, we discover *L*'s ultimate destination class one level at a time. If at any point we encounter an undecided class, we conservatively disable *P*.

In §5.5.4, we described how processing order does not always minimize the number of undecided classes encountered during testing; the same limitation

applies here. In §5.7, we define the *ultimate()* method that uses the same traversal of nesting levels described above to return ultimate destination class names.

5.6 Transforming Code

When superclass *P* is merged into its direct subclass *C*, the bytecode in *P*'s class file is moved to *C*'s class file and *P* is deleted. In this section, we describe the code transformations performed when bytecode is relocated and when classes are eliminated. We also describe how methods are inlined in merged classes.

Class merging relocates bytecode definitions that represent fields, methods, nested types, initializers, and constructors. Aside from the access control considerations discussed in §5.7, three important tasks accompany bytecode motion. First, all relocated constructors and any other relocated definitions that conflict with existing subclass definitions have to be renamed. Second, all references to relocated or eliminated definitions have to be updated. Third, methods are sometimes inlined instead of, or in addition to, being moved.

To perform these tasks, we first collect analysis information and create the *InRefs* set. *InRefs* is the set of all references *from* participating package types *to* (1) constructors and members of enabled classes, and (2) enabled top-level classes. *InRefs* is constructed using the class file information of participating package types (§5.2.3). Whenever we move, rename, or eliminate a bytecode definition, we use *InRefs* to locate all code that needs to reflect that change. In many cases, references are updated to point to a new bytecode definition.

The following subsections describe how definitions are relocated, how classes are eliminated, and how methods are inlined. These activities occur in the context of the class merging algorithm in §5.4.4. In particular, merging occurs from leaf to root in the fragments of *H'*. In our discussion, we use classes *P* and *C* as defined above.

5.6.1 RELOCATING DEFINITIONS

In this section, we describe how different types of bytecode definitions are moved from *P* to *C* and when they need to be renamed. Unless otherwise specified, references are always updated to point to the relocated bytecode using the *InRefs* set described above. In general, these references can reside in any participating package type.

We begin with *P*'s initializers, which are renamed if necessary to avoid name conflicts when they are moved to *C*. Initializers are not part of the class signature, so renaming has no effect on *C*'s signature. Implementations need to combine or chain together the bytecode for the special `<clinit>` initializer method called by the JVM (JVMS §3.9). Care must also be taken to order the execution of initialization code in the merged class to match that of the original classes (JLS §12.4). After merging, all references to *P*'s initializers reside in *C* and are updated appropriately.

P's constructors are converted into uniquely named private methods in *C* when they are relocated. *P*'s constructors are not inherited by *C*, so renaming has no effect on *C*'s signature. The Multiple Subclass (§5.3.7) and Explicit Allocation (§5.3.6) disabling conditions guarantee that the only calls to *P*'s constructors

are from constructors in *P* or *C*. After merging, all references reside in *C* and are updated appropriately.

Members of *P* that are inherited by *C* are moved into *C* without renaming. All references to these members are updated to point to the relocated bytecode.

Private fields, private nested types, and private static methods in *P* are renamed to avoid name conflicts when moved into *C*. After merging, all references to these private definitions reside in *C* and are updated appropriately.

Fields, nested types, and static methods in *P* that are hidden by members in *C* are renamed to avoid name conflicts when moved into *C*. These hidden members of *P* are not inherited by *C* (JLS §6.4.2), so renaming them can only widen *C*'s signature. Since *P*'s hidden members could be accessed using fully qualified names or the **super** keyword, references outside of *C* may need to be updated.

Instance methods in *P* that are overridden in *C* are not inherited by *C* (JLS §6.4.2), so they can be renamed. Since instance method lookup is dynamic, we update virtual calls to *P*'s overridden methods to refer to *C*'s overriding methods, not to *P*'s relocated methods. This type of reference update differs from all other updates because it does not point to the relocated bytecode from *P*, but instead to bytecode that already exists in *C*.

On the other hand, non-virtual calls to *P*'s overridden methods are updated to point to *P*'s relocated methods. These non-virtual calls use the **super** keyword and only reside in *C*. In addition, *P*'s private instance methods are moved to *C*

and renamed if necessary (JLS §8.4.6.3). Reference updates to these private methods also point to the relocated bytecode.

Figure 57 and Figure 58 illustrate the how references to instance methods are updated when the methods are relocated. In both figures, class `P` is merged into its child class `L`. Before optimization, participating package class `X` makes a virtual call to method `P.m()`. After optimization, relocated code is shown in bold font and method references are adjusted.

In Figure 57, `L` inherits `P.m()` before optimization. After optimization, `P` is eliminated and the reference to `m()` in `X` now points to `L.m()`.

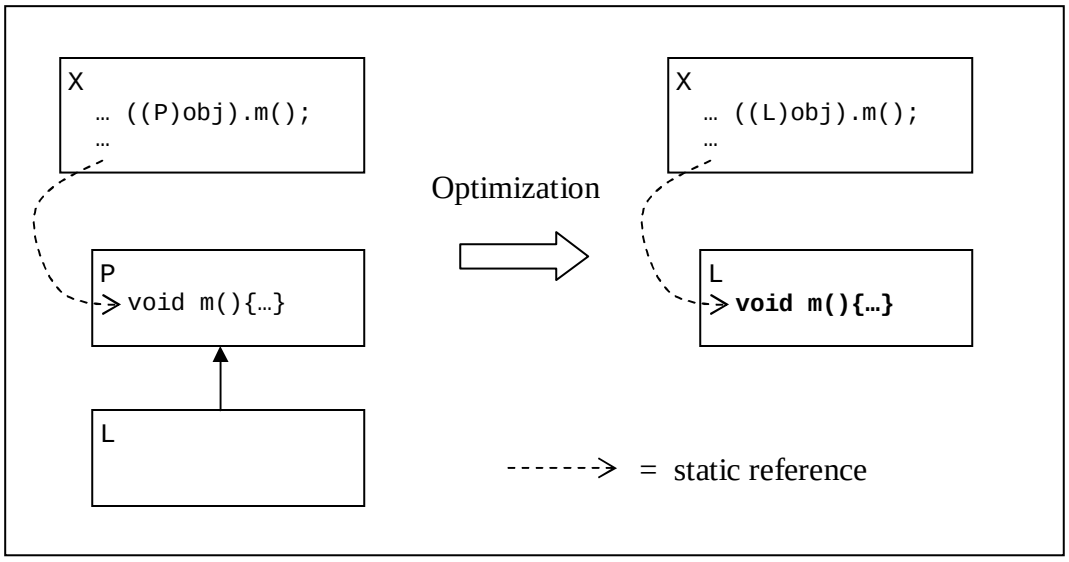


Figure 57 - Updating References for Non-Static Inherited Methods

In Figure 58, however, `L.m()` overrides `P.m()` before optimization. After optimization, `P.m()` is relocated to `L` and renamed `m'()`. The reference to `m()` in `X` does not point to this relocated code as it did in the previous example.

Instead, `X` now references `L.m()`, which is `L`'s original overriding method. In addition, `L`'s non-virtual call to `super.m()` references `m'()` after optimization, `P`'s relocated code.

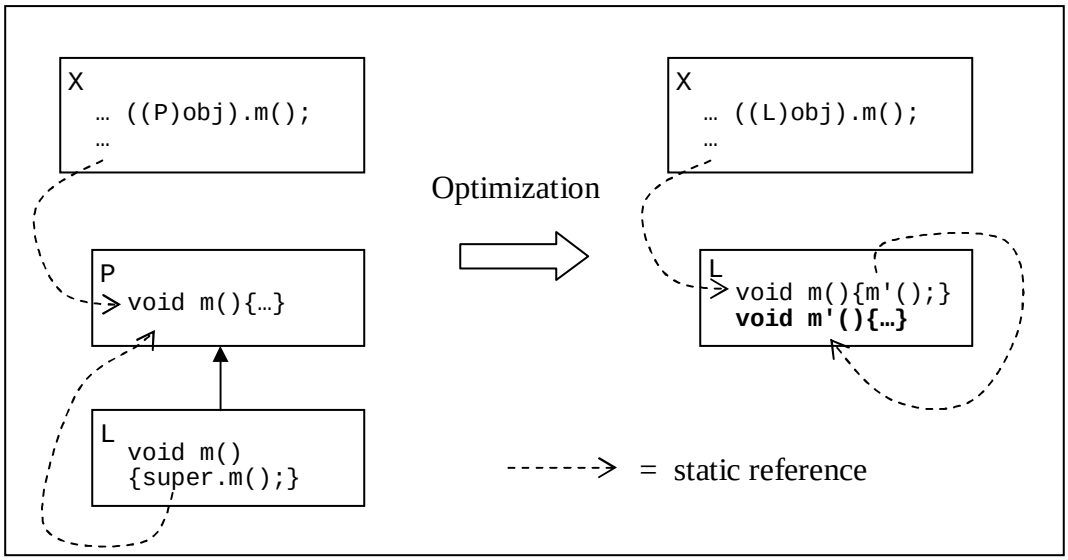


Figure 58 - Updating References for Overridden Methods

We have described how each type of bytecode definition is relocated and how references to them are updated. Before concluding our discussion, we note that all relocated constructors and all relocated overridden instance methods can be made private in *C*. We explained above why this is true for constructors, we now discuss the privatization of overridden instance methods. Before optimization, overridden instance methods can be either called non-virtually from *C* using **super** or called not at all because of dynamic dispatch. Thus, when these methods are relocated to *C* and made private, no access violations can occur. The benefit of making relocated constructors and overridden instance methods private is that

the *invokespecial* bytecode can continue to be used at *C*'s call sites (JVMS §7.7). Keeping *invokespecial* requires less code transformation and results in less runtime overhead. As a final note, implementations are free to delete relocated methods that are private and not referenced.

5.6.2 ELIMINATING CLASSES

After *P* is merged into *C*, *P* is deleted. Using the *InRefs* set, all references to *P* in any participating package type are updated to point to *C*. The Multiple Subclass disabling condition guarantees that *P* is superclass only to *C*. The Explicit Allocation disabling condition guarantees that *P* is never explicitly instantiated. In all other expressions that *P* can appear in, *C* can be safely substituted because *C*'s signature is a superset of *P*'s signature. Access control adjustment is considered in §5.7.

5.6.3 INLINING METHODS

In this section, we describe minimal inlining policies that address the problem of excessive method call overhead described in §2.4.4. We encourage implementations to explore other policies that can increase the overall effectiveness of inlining [2,9,61,81].

The simplest inlining policy that decreases method indirection in mixin-generated code is a policy that inlines constructors and overridden instance methods that have been merged from *P* into *C*. Recall that these relocated definitions can only be called from *C* because they are private (§5.6.1). If only one such call site exists, then inlining can even reduce *C*'s overall size because the private

method can be deleted. If multiple call sites exist, then a heuristic that considers inlined code size and the number of call sites should be used.

A more aggressive inlining policy could consider all private methods in *C* as candidates for inlining, not just those that were originally constructors or overridden instance methods in *P*. This more aggressive policy, as well as other alternative policies, can be carried out when *P* is merged into *C*. In addition, our design does not prohibit policies that take place at other times during optimization or policies that consider more than two classes at once.

5.7 Adjusting Access Control

When superclass *P* is merged into its subclass *C*, the bytecode in *P*'s class file is moved to *C*'s class file and *P* is deleted. In the previous section, we described how bytecode is transformed during class merging in all respects except those related to access control. We now complete our discussion on bytecode transformation by presenting algorithms that guarantee that references to and from relocated bytecode have access to their target declarations.

Before beginning our discussion, we introduce working definitions of the four types of access control available to Java declarations (JLS §6.6):

- **private** – access only from within the body of the top-level class that encloses the declaration.
- **package** – access from types in the same package (default access).
- **protected** – package access plus access from subclasses.

- **public** – access from all types without restriction.

Access control is adjusted in the context of the class merging algorithm in §5.4.4. When references are updated during bytecode relocation (§5.6.1) or class elimination (§5.6.2), the target declarations may need their access control adjusted. When *P* and *C* are in the same package, references remain valid without any changes to access control. When *P* and *C* are in different packages, however, references to declarations with package or protected access can be invalidated by class merging.

Figure 59 shows one way that references *to* relocated bytecode can be invalidated. The hierarchy shown has three classes that span two packages, `R` and `S`. Field `f` has package accessibility in class `R.Z` and is referenced from class `R.X`. The class hierarchy optimization merges `R.Z` into `S.Y`. After optimization, field `f` is in `S.Y` and the reference to `f` is still in `R.X`. This reference is now invalid because `R.X` does not have package access to `S.Y`.

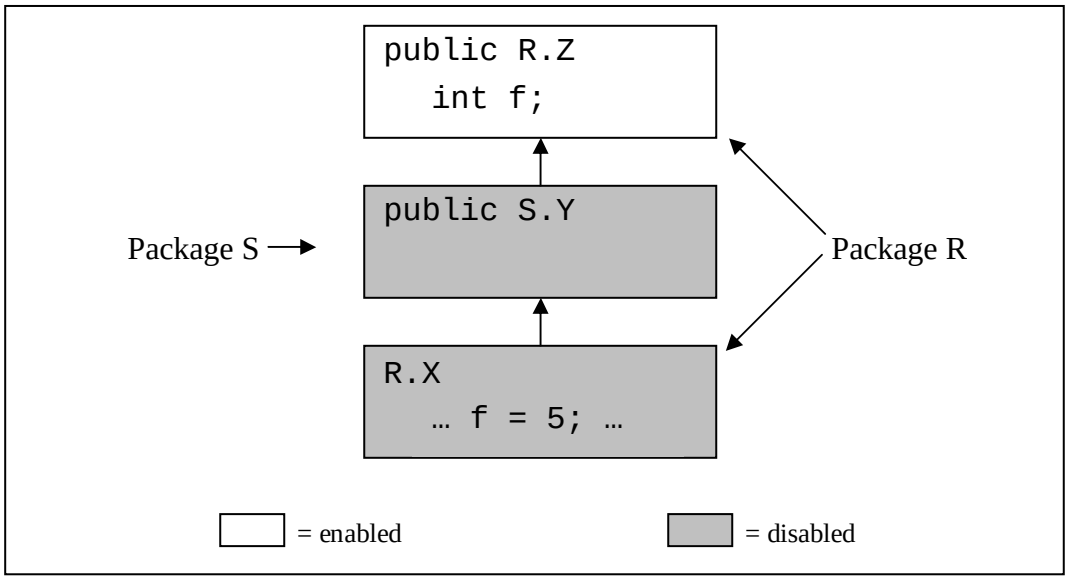


Figure 59 - References to Relocated Bytecode

Figure 60 shows one way that references *from* relocated bytecode can be invalidated. Optimization merges `R.Y` into `S.X` and moves the reference to field `f` out of package `R` and into package `S`. This reference is now invalid because `f` is only accessible inside package `R`. Note that even if `f` were made public in `R.Z`, the field could only be accessed from `S.X` if class `R.Z` were also made public.

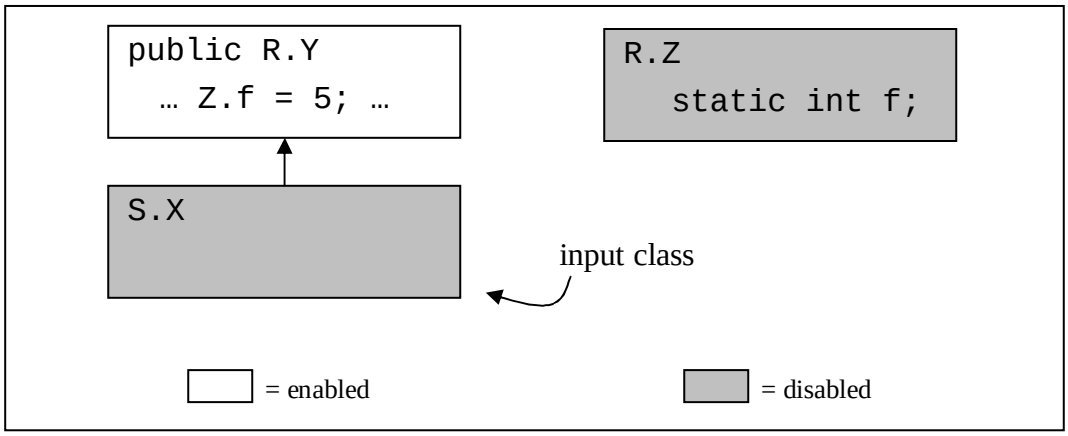


Figure 60 - References from Relocated Bytecode

Access control problems occur when bytecode contains or uses package or protected access control. When the bytecode is relocated across package boundaries, previously valid access patterns can be invalidated. These access problems can be avoided by (1) making public all relocated package and protected declarations, and by (2) making public all package and protected declarations referenced from relocated bytecode.

This *public promotion* approach to access control is simple to use and to implement. The main drawback of public promotion is that it weakens modularity. Classes that were crafted as abstract data types lose much of their data hiding capability and expose their internal representation to all other classes. Moreover, public promotion is imprecise because all declarations are changed, even those that are not actually referenced across packages.

To avoid the deficiencies of public promotion, we use a *minimal promotion* approach to access control. Our approach weakens access control on declara-

tions as little as necessary to guarantee that all existing references remain valid after optimization. For example, if declarations with package accessibility are moved between packages and all references remain valid, then access control is either not changed or made more restrictive. This selective adjustment is based on the needs of actual references, which makes minimal promotion more precise than public promotion.

In general, a promotion algorithm is *dynamic* if existing references are used to determine how access control is adjusted. A promotion algorithm is *static* if it does not consider existing references. Public promotion is static. Depending on the specific bytecode being relocated, we will see that minimal promotion is sometimes static and sometimes dynamic.

We use analysis information and special methods to execute the dynamic parts of the minimal promotion algorithm. We use the *InRefs* set defined in §5.6 to discover the references to relocated declarations and enabled top-level classes. In addition, we define the *ultimate()* method that returns the names of the ultimate destination classes in *H'*. This method runs only after *H'* has been constructed and all participating package types are either enabled or disabled.

To demonstrate its plausibility, we now sketch the implementation of the *ultimate()* method. *A.ultimate()* returns the name of the ultimate destination class for class *A*. *A.ultimate()* calls *A.leaf()* (§5.5.4.1) and assigns the result to *L*. If *L* is a top level class, then we append *L* to its package name and return the fully-qualified name. Otherwise, *L* is a nested class and its immediate enclosing class is *E*. We make a recursive call to *E.ultimate()*. The results from *E.ultimate()* are

combined with *L* to construct the fully qualified name of the ultimate destination class, which is returned.

We present the minimal promotion algorithm in two parts. The first part considers the static aspect of minimal promotion. The algorithm for static access control consists of rules that adjust control based solely on the bytecode definitions that are moved. The second part of our discussion considers the dynamic aspect of the minimal promotion algorithm. We identify four dynamic cases in which reference data must be consulted and we define a procedure to handle each case. Throughout our discussion, we use classes *P* and *C* as defined above.

5.7.1 STATIC ACCESS CONTROL RULES

This section describes the static part of the minimal promotion algorithm. Access control can be determined statically when it does not depend on reference patterns in the code. If *P* and *C* are in the same package, then package and protected declarations keep their current access control. All existing references into and out of relocated bytecode maintain valid access after class merging because accessibility has not changed.

On the other hand, if *P* and *C* are in different packages, then the problems of reference invalidation described above can occur. These cross package merging problems are addressed using dynamic algorithms, which we present in the next section. There are, however, five cases in which access control can be statically determined even if *P* and *C* are in different packages. These cases are as follows:

1. Private declarations in *P* remain private when they are moved to *C*. References to these private declarations remain valid after relocation because both the references and the declarations they point to are moved into *C*. Initializers are treated as private declarations for access control purposes.
2. In §5.6.1, we described how constructors in *P* are always made into private methods in *C*. This is a special case of statically determined access control.
3. Also in §5.6.1, we describe how overridden instance methods in *P* are made into private methods in *C*. This is another special case of statically determined access control.
4. Public declarations in *P* remain public when they are moved to *C*, except for the constructors and overridden instance methods that we just mentioned. References to public declarations maintain valid access because the declarations are still public in *C*.
5. References from relocated bytecode to public declarations do not require access control adjustment because the target declarations are still accessible from the bytecode's new location. These referenced public declarations can reside in any participating package type.

The five static cases listed above describe much of the access control processing that takes place during class merging; we now describe the remaining dynamic cases.

5.7.2 DYNAMIC ACCESS CONTROL RULES

This section describes the dynamic part of the minimal promotion algorithm. Access control is determined dynamically when it depends on reference patterns in the code. When *P* and *C* are in different packages, then the problem of reference invalidation described in §5.7 can occur. We identify four cases where static access control assignment is not sufficient and where reference data must be consulted. We then define a procedure to handle each of these cases.

Each dynamic procedure updates access control on a specific class of definitions. The first procedure updates access control on relocated package declarations; the second procedure updates relocated protected declarations. The third procedure updates classes that have their superclasses merged into them. The fourth procedure updates declarations that are referenced from relocated code, but are not themselves relocated.

Each dynamic procedure is described in it own subsection. The procedures use the *InRefs* set and the *ultimate()* method as described in §5.7. In each subsection, our overall goal is to merge superclass *R.P* into its subclass *S.C*, where *R* and *S* represent the different packages in which classes *P* and *C* reside. We define the ultimate destination class for all relocated members as follows:

$$UltimatePkg.UltimateClass = R.P.ultimate()$$

The package of *R.P*'s ultimate destination class is represented by *UltimatePkg*; the class is represented by *UltimateClass*. We now describe the four dynamic procedures.

5.7.2.1 Adjusting Relocated Package Declarations

The procedure below adjusts access control on non-overridden, relocated members that have package access. For each member *m* with package access in *R.P* that is not overridden in *S.C*, we perform the following:

1. If any reference to *m* in *InRefs ultimately* resides in a package different from *UltimatePkg*, then assign *m* public access.
2. Otherwise, if *UltimatePkg* equals *R*, or if any reference to *m* in *InRefs ultimately* resides in a type *T* different from *UltimateClass*, then assign *m* package access.
3. Otherwise, assign *m* private access.

The above procedure compares the ultimate destination of non-overridden, relocated package members to the ultimate destination of their references. In step 1, package members are made public if relocation invalidates any reference. In step 2, package members keep their package access control if they are ultimately referenced from classes in *UltimatePkg* other than *UltimateClass*.[16]

Interestingly, in step 3, package members are made private if they are not ultimately referenced from outside of *UltimateClass*. This *strengthening* of ac-

[16] Step 2 can be fine tuned to *not* handle cases in which *T* and *UltimateClass* share a common enclosing type, but this is a minor adjustment.

cess control is justified because programmer intent to expose members within their original package does not automatically translate into intent to expose them within a different package after relocation. Note that if the original and ultimate packages for *P* are same, access control is never strengthened.

5.7.2.2 Adjusting Relocated Protected Declarations

The procedure below adjusts access control on non-overridden, relocated members that have protected access. For each member *m* with protected access in *R.P* that is not overridden in *S.C*, we perform the following:

1. Define $InRefs_m$ as the set that contains the ultimate destination classes in which all references to *m* in *InRefs* reside.
2. If there exists a $p_m.c_m \in InRefs_m$ such that (1) p_m is different from *UltimatePkg* and (2) c_m is not a subclass of *UltimateClass*, then assign *m* public access.
3. Otherwise, assign *m* protected access.

The above procedure compares the ultimate destination of non-overridden, relocated protected members to the ultimate destination of their references. If a reference ultimately resides in a class that is not a subclass of and is not in the same package as *UltimatePkg.UltimateClass*, then the referenced member is made public. Otherwise, the member's access control remains protected.

5.7.2.3 Adjusting Merged Classes

The procedure below adjusts access control on subclasses that absorb their superclasses, which are then eliminated. References to these superclasses are changed into references to the merged subclasses. Our access control procedure executes one of two cases depending on whether the ultimate destination class is top-level or nested.

Case 1: If *UltimatePkg.UltimateClass* is a top-level class, then for each type *T* that contains a reference to *R.P*, we let T_u = *T.ultimate()*. We then perform the following test:

> If *UltimatePkg.UltimateClass* has package access and T_u resides in a different package than *UltimatePkg*, then we assign *UltimatePkg.UltimateClass* public access.

Case 2: If *UltimatePkg.UltimateClass* is a nested class, then for each type *T* that contains a reference to *R.P*, we let T_u = *T.ultimate()*. We then perform the following test:

1. If *UltimatePkg.UltimateClass* is public, no adjustment is necessary and the test terminates.

2. Otherwise, if the package that T_u resides in is different than *UltimatePkg*, and T_u is not a subclass of *UltimatePkg.UltimateClass*, then we assign *UltimatePkg.UltimateClass* public access.
3. Otherwise, if the package that T_u resides in is different than *UltimatePkg*, and T_u is a subclass of *UltimatePkg.UltimateClass*, then we assign *UltimatePkg.UltimateClass* protected access.
4. Otherwise, if *UltimatePkg.UltimateClass* is private and if T_u and *UltimatePkg.UltimateClass* do not share a common enclosing type, then we assign *UltimatePkg.UltimateClass* package access.

The procedures in the above two cases terminate as soon as *UltimatePkg.UltimateClass* has public access control. All adjustments represent the minimal weakening of access control necessary to maintain the validity of existing references. In Case 1, top-level types can only have public or package access, so only one adjustment is possible.

In Case 2, nested classes can have all four levels of accessibility, so three adjustments are possible. Steps 2 and 3 cover all configurations in which T_u and *UltimatePkg.UltimateClass* are in different packages; step 4 covers the case when they are in the same package.

5.7.2.4 Adjusting Stationary Declarations

The previous three dynamic procedures adjust access control on relocated code. The procedure in this section adjusts access control on package and pro-

tected declarations that are not moved during optimization, but that are referenced from bytecode that is moved. We begin by defining terms used in the procedure.

Declarations not moved by our optimization are called *stationary declarations*. These declarations can be constructors, members or top-level types. We detect stationary declarations by first identifying *stationary types*. Stationary types are participating package types that are disabled and, if they are nested, have only disabled enclosing types. A declaration is stationary if and only if the declaration either is a stationary type or resides in a stationary type.

Our procedure processes each declaration referenced in *R.P*'s bytecode. For each declaration *d* pointed to by some reference in *R.P*'s bytecode, we perform the following:

1. If *d* is public or private, then *d* is not changed.
2. If *d* is not a stationary declaration, then *d* is not changed.
3. If *d* resides in package *UltimatePkg*, then *d* is not changed.
4. If *d* is a protected declaration that resides in a superclass of *UltimatePkg.UltimateClass*, then *d* is not changed.
5. Otherwise, we assign *d* public access.

The above procedure changes *d*'s access control to public only when tests 1-4 fail. Test 1 eliminates declarations handled by our static procedure. Test 2 eliminates declarations handle by other dynamic procedures. Tests 3 and 4 eliminate package and protected stationary declarations that can still be referenced

from *R.P*'s relocated bytecode. If no test is satisfied, the stationary declaration is made public.

5.8 Discussion

The class hierarchy optimization takes inlining a step further by merging classes as well as methods. Class merging and method inlining are both repackaging techniques that move code to improve performance. Both transformations preserve program semantics by executing existing code, though the location and context of that code is changed.

We started with the simple idea of flattening Java class hierarchies and then worked through the details of its high-level design. We found that determining when optimization can be applied is as difficult as actually performing the optimization. We also found that handling Java's nested types, package system, and access control semantics represent the most challenging aspects of the design. If there is a lesson here, it is that simply expressed type transformations are not necessarily simple to implement in a real language. Recent work in *refactoring* [122] supports this conclusion.

Our design establishes the basic concepts, transformations, and data structures used in the class hierarchy optimization. There are, however, many possible variations that implementations may choose to explore. We list some of these alternatives here:

1. Nested hierarchies with private leaf classes can be included in the set of associated nested hierarchies so that they can also be optimized (§5.1.8).
2. The immediacy requirement in the Lexical Nesting condition can be relaxed to increase the number of optimizable classes (§5.3.8).
3. Alternatives to the outside-in processing order (§5.4.3.1) can be explored to more precisely determine when classes should be disabled (§5.5.4).
4. Multiple access control adjustment strategies can be supported, including public promotion and variations on minimal promotion (§5.7).
5. Certain disabling conditions, such as merging static and non-static classes (§5.3.5), can be relaxed by inspecting the actual classes being optimized.
6. Interface merging can be explored.

5.9 Related Work

This section discusses three areas of work related to JL's class hierarchy optimization. We describe how our optimization relates to an application compression tool, to refactoring tools, and to other optimizations that also depend on class hierarchy structure.

5.9.1 THE JAVA APPLICATION EXTRACTOR (JAX)

The Java Application Extractor (Jax) [121] implements a number of transformations that reduce the size of Java programs. These transformations remove

unused classes, methods and fields from programs. To reduce program size, Jax can eliminate a class after merging it into its superclass. Jax does not, however, merge two classes if runtime objects become larger as a result of the merge. This restriction implies that if a class contains live, non-static fields, then the class cannot be merged into its superclass. In addition, Jax does not merge a class and its superclass if a live, non-abstract method *m* is declared in both classes.

The goal of JL's class hierarchy optimization is not to reduce program size, but to merge as many classes as possible in a set of related hierarchies. In JL, classes with live, non-static fields are merged. In addition, JL can merge two classes that both contain a live, non-abstract method *m*. On the other hand, JL has a number of disabling conditions, such as Explicit Allocation (§5.3.6) and Multiple Subclasses (§5.3.7), that restrict merging in ways that Jax does not.

Jax and JL also differ in a number of minor ways. Jax merges interfaces into classes, while JL does not. Also, Jax uses public promotion to handle cross-package access control in merged code. JL specifies the minimal promotion algorithm, but encourages implementations to offer multiple approaches to access control, including public promotion. Both Jax and JL perform method inlining, but Jax inlines methods only if program size does not increase.

Lastly, Jax and JL share a number of common traits. Both optimizations distinguish between application classes that can be optimized and system classes that are never modified. In addition, JL adopts Jax's approach to managing reflection and dynamic loading. Both optimizations rely on programmers to charac-

terize the safe and unsafe uses of these facilities. In Jax, programmers specify the unsafe uses; in JL, programmers specify the safe uses.

5.9.2 REFACTORING

Object-oriented classes are *refactored* [75,80,87,94,122] when their structure is changed in behavior-preserving ways. Refactoring includes the creation, deletion, and renaming of classes, methods and fields; it also includes changing a class's superclass. The class hierarchy optimization refactors Java code, but it differs from most refactoring tools in intent. The goal of our optimization is to create efficient code, whereas the goal of refactoring tools is to improve program design.

These different goals lead to differences in technology and usage. The class hierarchy optimization processes bytecode and applies the same transformation to all input. In addition, the optimization runs with as little programmer interaction as possible. Refactoring tools, on the other hand, process source code and their output is expected to be edited by programmers. Moreover, refactoring tools are often implemented as browsers so that programs can be restructured interactively.

5.9.3 DETERMINING CLASS HIERARCHY STRUCTURE

The class hierarchy optimization uses the closed world assumption (§5.2.1) to limit the classes it examines. This need to define a limited set of classes at compile-time is not unique to our optimization. For example, the JOVE™ [60] and TowerJ™ [123] native Java compilers support optimizations

that require static access to all classes that can be encountered at runtime, including classes dynamically loaded by the application.

Since the release of Java 1.2, however, packages can be *sealed* [115] in Java archive (JAR) files [8]. A sealed package is one in which all classes in the package must be loaded from the same JAR file. Zaks, Feldman and Aizikowiz [131] have shown that sealed packages can be used to increase the number of statically devirtualized method calls. More specifically, since package classes in sealed packages have no visibility outside the package, and since no new classes can be added to a sealed package, sealed packages provide enhanced information for resolving method calls at compile-time. As a possible subject for future work, the class hierarchy optimization could use sealed packages to reduce its reliance on the closed world assumption.

CHAPTER 6 COMPILER IMPLEMENTATION

This chapter describes the implementation of Java Layers. We actually describe the implementation of two different versions of the language. The first version of Java Layers (JL1) served as a prototype and test bed for exploring ideas on software construction. JL1 explicitly implements the GenVoca model (§2.3.1) and was used to perform our ACE evaluation (§7.1). JL1 is presented in §6.1.

The second version of Java Layers (JL) builds on lessons learned designing and implementing JL1 (§6.1.1); this is the version of Java Layers described throughout this dissertation. The distinguishing characteristic of JL is its focus on integration with current programming languages. Our emphasis on integration is practical: Mixins extend common object-oriented programming techniques, so to reach the widest audience, mixin implementations should extend common object-oriented languages. JL's implementation is presented in §6.2.

In this chapter, we also explore an interesting implementation problem that highlights how feature interaction makes language design a subtle art. The problem involves naming JL's instantiated types. JL's heterogeneous implementation (§4.1.4.1) specializes code for each distinct instantiation of a parametric type, and each of these specializations must be uniquely named. In §6.3, we describe a name mangling technique that accounts for Java's package system and for JL's implicit **This** type parameter (§4.3).

6.1 Direct Implementation

The first version of Java Layers (JL1) [30] reflects three design goals. The first goal is to implement the main elements of the GenVoca model (§2.3.1) by supporting the concepts of layers, realms and type equations. This direct support of the GenVoca model allows programmers to create and compose software components; it also allows applications to be built using the methodology of stepwise program refinement. The second goal is to implement language features that make component programming easier. These features are the precursors to the JL features described in Chapter 4. The third goal is to localize our modifications to Java primarily in the new *layer* construct. This approach compartmentalizes JL1's implementation and allows programmers to work in standard Java until they are comfortable with component programming.

JL1 is implemented as a source-to-source compiler that transforms JL1 source code into Java source code and then invokes a Java compiler to produce bytecode. The JL1 compiler is approximately 16K lines of Java code, not counting grammar files or test code.

In JL1, GenVoca components are defined using the layer construct. Layers belong to one or more realms, which are defined as Java interfaces. Layers in the same realm can be substituted for one another. Type equations are used to combine layers, Java classes, and Java interfaces into applications. The inputs and output of the JL1 compiler are shown in Figure 61.

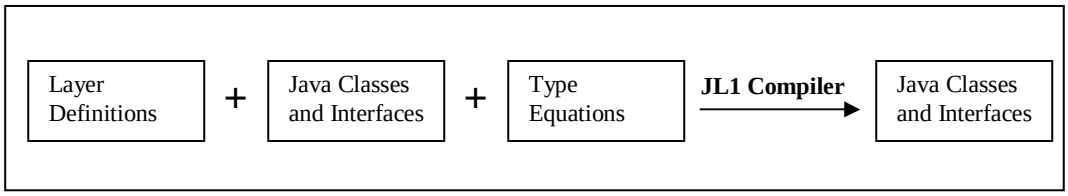


Figure 61 - Compilation in JL1

Figure 62 uses JL1 layers to redefine the transport example introduced in §2.4.3 on page 25. The `TransportIfc` interface defines the transport realm, which is populated by classes that implement `TransportIfc` and layers that export `TransportIfc`. The figure shows two realm members, the `TCP` layer and the `Secure` layer. Both of these layers are transformed by the JL1 compiler into Java classes that implement `TransportIfc`. The `Secure` layer declares constrained type parameter `T`, which the **mixin** keyword designates as the placeholder for the superclass of the class generated by `Secure`. Figure 62 defines a type equation that generates a hierarchy of three classes. The leaf class, `Trans`, implements a secure TCP transport class. The superclass of `Trans` is the class generated by `Secure`, and the root of the hierarchy is the class generated by `TCP`.

```
public interface TransportIfc {     // Realm
 public void send(byte[] out);
 public void recv(byte[] in);
 public void disconnect();
}

layer TCP exports TransportIfc {
 public void send(byte[] out){…}
 public void recv(byte[] in){…}
 public void disconnect(){…}
}

layer Secure<mixin T implements TransportIfc>
 exports TransportIfc
{
 public void send(byte[] out){…}
 public void recv(byte[] in){…}
 public void disconnect(){…}
}

class Trans = Secure[TCP];          // Type equation
```

Figure 62 - Defining Software Components in JL1

Figure 62 illustrates how JL1 layers support their own form of mixin inheritance. JL1 also defines its own version of deep conformance (§4.2), a most derived type expression (§4.3), constructor propagation (§4.4) and semantic checking (§4.5). In JL1, these features are only used inside layer constructs.

We now describe what we learned from JL1 and how we incorporated that knowledge into the second version of Java Layers.

6.1.1 LESSONS FROM DIRECT IMPLEMENTATION

One of the most important lessons that we learned from implementing JL1 is that the benefits of generic programming should be available in all types, not

just in the special layer construct. JL1 is specialized to support stepwise program refinement, which is a developing software engineering methodology. JL1 does not, however, provide parametric classes and interfaces that conveniently support traditional generic programming. This limitation makes JL1 useful for research, but less desirable as a general purpose programming language.

In the second version of Java Layers (JL), we eliminate JL1's layer construct by implementing mixins in classes and interfaces. This revised implementation supports both mixin programming and traditional generic programming. JL also supports deep conformance and constructor propagation in both parametric and non-parametric types. In addition, certain JL1 capabilities, such as automatic interface generation, proved not to be useful in practice and are not included in JL. Similarly, certain capabilities missing from JL1 are included in JL. For example, the occasional need to selectively apply deep conformance to non-public nested types led us to include the **propagate** keyword in JL (§4.2.3).

6.2 Integrated Implementation

The main impetus for designing a second version of Java Layers (JL) was to show that mixins integrate well with generics in Java. In §4.1.4, we list recent proposals for adding parametric polymorphism to Java. These proposals, along with Thorup's proposal (§4.3.5) for adding virtual types to Java, represent the latest thinking on adding generics to Java. JL's generic types share much of the syntax, semantics, and capabilities of these other proposals. JL, however, is the only

compiler of Java generics that implements mixins, and JL's extended support for mixin programming distinguishes it further.

There are two important benefits in designing JL as an integrated generic solution. First, JL's parametric classes and interfaces can be used as traditional generics, which means JL reaches a wide audience before mixin programming is even considered. Second, JL is *practical* to implement. Languages that already support parametric types can support mixins with modest changes to their grammars—our implementation of JL proves this. Conversely, object-oriented languages that support mixins can easily support non-mixin parametric types as well.

JL represents a research platform in which the interaction between mixin and non-mixin generic programming can be studied. Our ultimate goal is to integrate into mainstream languages what we learn about mixin programming using JL, just as F-bounded polymorphism [28] and constrained type parameters [29,109] have been integrated into languages.

JL is implemented as a source-to-source compiler that transforms JL source code into Java source code and then invokes a Java compiler to produce bytecode. The JL compiler is approximately 18K lines of Java code, not counting grammar files or hundreds of test cases.

The current implementation of the JL compiler does not completely implement the JL language. We have already mentioned that the semantic checking facility is not implemented (§4.5). In addition, the compiler does not write extended attributes to class files, which leads to other limitations. For example, type parameter bindings, instantiation expressions, and the propagate attribute on con-

structors are not saved in bytecode, so this information is only available when source code is loaded. The release notes on JL's web site [57] lists the limitations of the current implementation.

One of the more interesting implementation challenges encountered when writing the JL compiler involved naming instantiated classes and interfaces, which we now describe.

6.3 Name Mangling in JL

In this section, we specify how instantiated parametric types are named in JL. In §4.1.4.1, we noted that JL's heterogeneous implementation requires that uniquely instantiated parametric types must be uniquely named. In §4.3.3.3 on page 85, we used a simplified naming scheme that incorporated the implicit **This** type parameter into generated names. Using that scheme, instantiation names include the parametric type name, a representation of the This-binding name, and the names of the types bound to all other type parameters.

The simplified naming scheme is similar to the name mangling [68] approach used in C++ template [110] instantiations. In C++, name mangling essentially concatenates the template name and, in declaration order, the names of the types that get bound to the type parameters.

There are two reasons why the name mangling used in C++ and the simple scheme used in §4.3.3.3 will not work in JL. First, the length of fully-qualified type names in Java not only makes them inconvenient to use, but possibly too long to fit in command-line buffers of some operating systems. Second, in mixin-

generated hierarchies, the This-binding of a parent instantiated type depends on the This-binding of its child, and the This-binding of a child indirectly depends on the This-binding of its parent. In §6.3.2.2, we describe how we break this circular dependency so that instantiations can be named.

The following subsections describe how JL's naming scheme addresses the above two issues. We first describe how JL bounds the length of instantiated type names using hashing. We then describe how JL provides special support for **This** during instantiation, including an algorithm that uses *name tokens* to break the circular dependency involving This-bindings.

6.3.1 HASHING

In Java, a type is identified by its fully-qualified name, which can include package and enclosing type components. For safety and precision, only fully-qualified names are used to name instantiations in JL. For example, if *String* is specified as an actual type parameter, then the name *java.lang.String* is used.

Fully-qualified type names in Java applications tend to be long. One factor that increases name length is Java's package naming convention [8]. For example, *PropagateRecord* is a nested class used in the JL compiler implementation. Using the recommended naming convention, its fully-qualified name is *edu.utexas.cs.jl.compiler.JLCConformer.PropagateRecord*. When used as a type parameter, *PropagateRecord* would significantly lengthen an instantiation's name if a simple name scheme like that in §4.3.3.3 were used.

The name length problem becomes acute, however, when mixins are used. To understand the problem, consider the declaration of `Leaf` in Figure 63. `Leaf`

inherits from a composition of mixin classes. All types shown are in the `edu.utexas.cs.myapp` package. Using a naming scheme similar to that in §4.3.3.3 on page 85, the name generated for the superclass of `Leaf` contains 228 characters. This naming scheme combines a parametric type name with all its actual type parameter names, including its **This** type name, to generate a unique instantiation name. Obviously, the use of longer names, more type parameters, or deeper mixin hierarchies only exacerbates the problem.

```
package edu.utexas.cs.myapp;
class Leaf
 extends MyType1<MyType2<MyType3<MyType4<BaseType>>>>
{…}
```

The Name Generated for Leaf's Superclass

```
MyType1<
  :edu.utexas.cs.myapp.Leaf, edu.utexas.cs.myapp.MyType2<
    :edu.utexas.cs.myapp.Leaf, edu.utexas.cs.myapp.MyType3<
      :edu.utexas.cs.myapp.Leaf, edu.utexas.cs.myapp.MyType4<
        :edu.utexas.cs.myapp.Leaf, edu.utexas.cs.myapp.BaseType>>>>
```

Figure 63 - Name Inflation using Mixins

The generation of type names with hundreds or even thousands of characters introduces usability concerns. In Java, the type name is also used as the class file name. Long file names are not only difficult for programmers to key in, but they are difficult to use if they exceed the operating system's command line input buffer, which can be as little as 260 bytes in current systems.[17] In addition, file systems and file utilities can impose their own limitations on file name lengths.

[17] Microsoft Windows NT™.

To avoid the usability problems of long type names, JL generates a unique string for each instantiated type and then uses a hash of that string to construct the actual instantiation name. We now describe this name generation algorithm, which is called recursively in instantiations that contain other instantiations. The algorithm begins by substituting fully-qualified names for all actual type parameter names in an instantiation. The This-binding name appears as the instantiation's first parameter and is preceded by a colon (:).

The algorithm then normalizes all type parameter literals and removes all white space to create a *canonical representation* of an instantiation expression. This canonical representation is a string, which is hashed to an unsigned 32 bit number. The instantiation name is then constructed by concatenating the parametric type name with an underscore character (_) and the hexadecimal hash number. The length of the generated name is no greater than the parametric type name plus nine characters. For example, the name generated by JL for the superclass of `Leaf` in Figure 63 is `MyType1_246332E`.

JL hashes strings using an adaptation [18] of Holub's hash function [52], which is based on Weinberger's generic hashing algorithm [2]. Since JL generates names using all of the 2^{32} possible values returned by the hash function, name collisions are unlikely to occur in practice.[18] For correctness, however, the JL compiler can check existing files before writing new instantiations to file. If files with the same name contain different types, then compilation aborts. This check

[18] We have never experienced a name collision.

is not currently performed because, as we note in §6.2, JL attributes like type parameter bindings are not written to Java class files.

6.3.2 NAME MANGLING WITH *THIS*

This section describes two issues involving instantiation and the implicit **This** type parameter. The first issue concerns the replication of nearly identical types during instantiation, which can happen when parametric types do not use **This**. The second issue concerns the circular dependency involving This-bindings that we mentioned above in §6.3.

6.3.2.1 Avoiding Duplicate Type Generation

In this section, we first describe how the implicit **This** type parameter can cause unnecessary code replication. To address this problem, we introduce the concept of *This-specialization* and use this concept to avoid generating unneeded code specializations. We then describe how This-specialization is implemented.

Figure 64 shows how nearly identical versions of a class can be generated during instantiation. The figure defines parametric class `Base<T>` and two non-parametric classes, `A` and `B`, which inherit from `Base<int>`. Also shown are two distinctly named instantiations of `Base<int>`. These two instantiations are generated because **This** is bound to `A` in one instantiation and to `B` in the other. As described in §6.3.1, the types bound to **This** are part of the string that gets hashed when naming an instantiation. Different instantiation names represent different instantiation types, thus two distinct hierarchies are generated by the code in Figure 64.

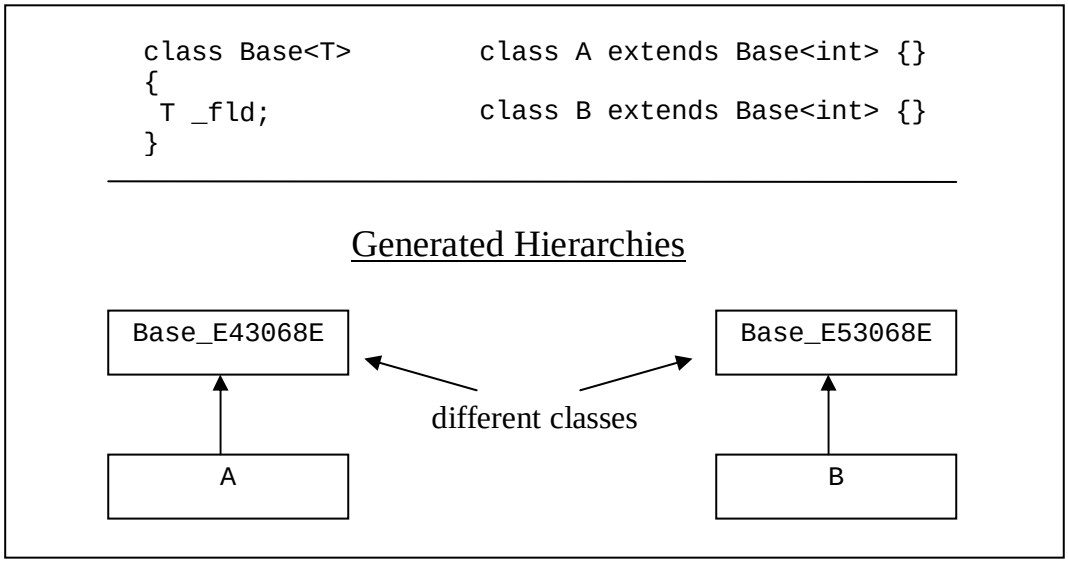


Figure 64 - Generating Identical Types with Different Names

The above example exposes two problems. First, memory is wasted when types that are different only in name are replicated during instantiation. In the above instantiations, the two versions of `Base<int>` differ only in their This-bindings, which are never used because **This** is never referenced. Second, since `A` and `B` do not share a common `Base<int>` supertype, objects of types `A` and `B` cannot be treated as the specializations of the same parametric type. This limits the expressiveness and flexibility of the language.

To avoid needless code replication and to generate hierarchies that more precisely reflect inheritance relationships, we introduce the concept of *This-specialization* in JL. A parametric type is *This-specialized* if its implementation depends on **This**. Instantiations of This-specialized types generate different byte-

code depending on their This-binding, so these instantiations must be uniquely named.

On the other hand, parametric types that are not This-specialized generate the bytecode that differs only in name no matter what type gets bound to **This**. Instantiations of these parametric types that differ only in their This-bindings can use the same name, which means that *we ignore **This** when it is not used*. For parametric types that are not This-specialized, instantiations are named as if **This** did not exist in JL, which solves the problem of replicated bytecode.

In terms of implementation, we need to distinguish between This-specialized types and those that are not This-specialized in order to correctly name instantiations. A parametric type *P* is This-specialized if (1) its implicit **This** type parameter is used or referenced or (2) any supertype of *P* is This-specialized. The first condition follows from the definition of This-specialization and can be checked by inspecting the source code of a parametric type.

The need for the second condition is demonstrated in Figure 65, which shows why subtypes inherit This-specialization from their supertypes. `Z<>` appears in two hierarchies and is This-specialized. In the hierarchy with leaf class `A`, **This** is bound to `A`; in the hierarchy with leaf class `B`, **This** is bound to `B`. These hierarchies require different instantiations of `Y<>` to inherit from different instantiations of `Z<>`. Since the only way to differentiate instantiations of `Y<>` is to use their This-bindings, `Y<>` must be This-specialized.

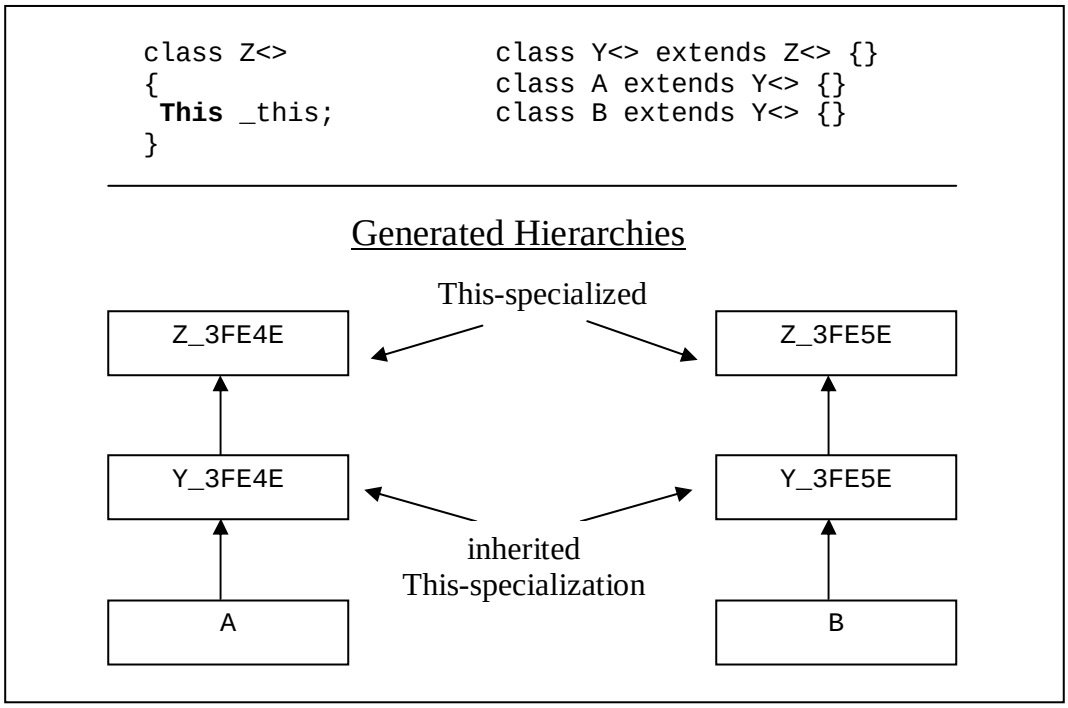


Figure 65 - Inheritance of This-Specialization

We can think of subtypes as inheriting This-specialization from their supertypes because This-specialized supertypes force their subtypes to also be This-specialized. (Of course, the lack of supertype This-specialization does not prevent a subtype from being This-specialized.) JL implements the supertype condition of This-specialization by querying supertypes during instantiation.

JL currently implements a conservative approximation of This-specialized naming. The implementation is conservative because it sometimes assumes that a supertype is This-specialized rather than perform an extensive query. This approach is easy to implement and provides most of the benefits of a precise implementation.

6.3.2.2 Avoiding Circular Dependencies during Name Generation

In §6.3, we noted that circular dependencies involving This-bindings are sometimes encountered when instantiations are named. In Figure 66, we use the classes `M<T>`, `Z<>`, and `M<Z<>>` to illustrate the problem.

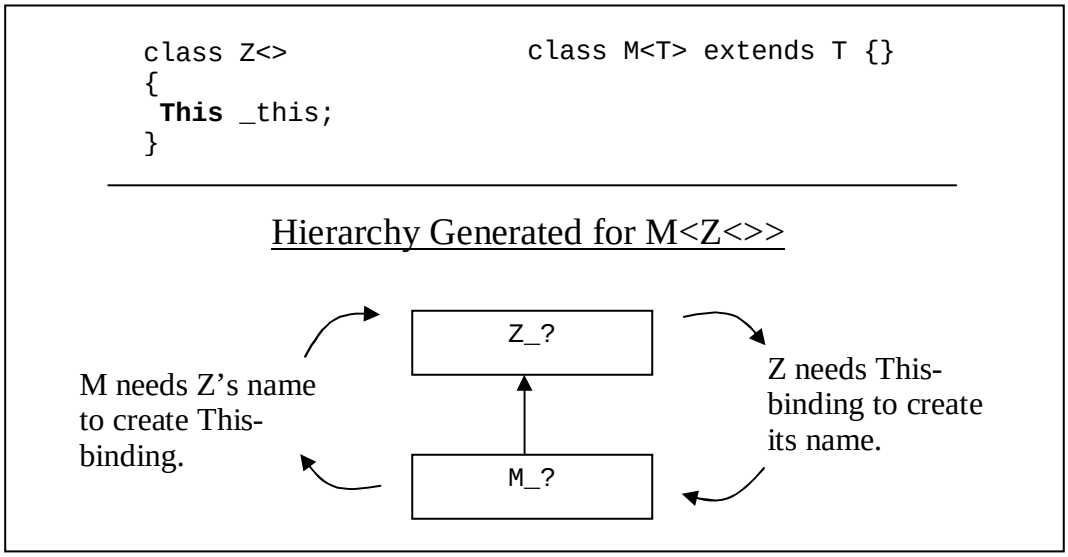


Figure 66 - Circular Dependency in Name Generation

Using the naming algorithm from §6.3.1, the leaf class of instantiation `M<Z<>>` will be named `M_?`, where the question mark represents a hash of `M<T>`'s parameters. `M<T>`'s only explicit parameter is `Z<>`, which is an instantiation of a This-specialized parametric type. To determine the value of the question mark in `M_?`, we need to generate the name of `Z<>` according to the following steps:

1. *Determine `Z<>`'s This-Binding.* The only type parameter binding needed to name an instantiation of `Z<>` is the This-binding. The This-binding rules for inheritance contexts (§4.3.3.2) specify that the This-binding of `Z<>` is the same as the This-binding of `M<Z<>>`.

2. *Determine `M<Z<>>`'s This-Binding.* The This-binding of `M<Z<>>` is determined by the rules for non-inheritance contexts (§4.3.3.1), which require the name of the instantiation `M<Z<>>`. That is, we need to know the value of the question mark in `M_?`, which the initial problem that we are trying to solve.

The circular dependency described above occurs because the name of `M<Z<>>` depends on the name of `Z<>` and vice versa. This circularity can only occur when This-specialized parametric types are parameters to mixins. Non-parametric leaf classes, such as `A` and `B` in Figure 65, bind **This** before instantiation begins, so the circularity problem is avoided. Circularity is also avoided if the This-specialized parameter is in a non-inheritance context (§4.3.3.1). In addition, circularity is also avoided if **This** is explicitly bound in the leaf type or if the leaf type is an interface (§4.3.3).

In cases where circularity does exist, JL uses a *name token* algorithm that splits name generation into two steps and breaks the circular dependency. The first step of the algorithm constructs an instantiation-related name token. This token is a string that is passed to supertypes as a temporary placeholder for the leaf's This-binding name. We construct the string token by concatenating the leaf class name and the names of all its parametric supertypes. Each name is followed

by an asterisk (*). We start concatenating at the instantiation's leaf and proceed towards the root until a non-parametric type is encountered. In Figure 66, for example, the token "`M*Z*`" is constructed for the instantiation `M<Z<>>`.

The second step of the algorithm binds **This** in the parametric leaf class. The leaf class first requests each actual type parameter to generate its own name. Parametric supertypes of the leaf class use the name token as a temporary This-binding to generate their names. This process continues recursively from leaf to root in the instantiation hierarchy until all names are generated. At this point, the leaf class has the information it needs to generate its own name. The leaf class generates its name using the name token and its type parameter names. The leaf class then binds its name to **This**.

From this point on, instantiation proceeds using the leaf class's This-binding as described in §4.3.3. Name tokens help guarantee that different instantiated types are given different names by encoding the order of types in hierarchies. If types are added, deleted or appear in a different order in a hierarchy, then the token will reflect the difference and cause dissimilar names to be generated.[19] Thus, different compositions of parametric types generate different names.

The name token algorithm, however, does have a side effect. In place of a This-binding name, the algorithm hashes a name token into the names of instantiated types. These instantiation names cannot be easily generated in other contexts because the name token is not a type name. For example, using the defini-

[19] JL currently uses only the leaf class name to create name tokens, which is less discerning than the algorithm described here.

tions in Figure 66, no implicit or explicit This-binding can be specified on the left-hand side of the assignment below that will allow compilation to succeed.

```
Z<:?> z = new M<Z<>>();  // Compile always fails.
```

The above code could compile if the string "`M*Z*`" were substituted for the question mark, but this substitution is not currently allowed in JL. The problem can be avoided by using non-parametric leaf classes, but the problem still represents a limitation in the expressiveness of the current version of JL.

6.3.2.3 Discussion

JL's implicit **This** type adds a parameter to all generic types and, in doing so, affects the generation of type names in JL's heterogeneous implementation of parametric polymorphism. In languages like Java, class and interface names are of central importance because they distinguish types. The two preceding sections describe how JL handles complications that **This** introduces into name generation.

The interaction between language features and name generation can be subtle in heterogeneous implementations. In particular, we saw how the naming limitation described at the end of the last section restricts the expressiveness of JL. To remove this restriction, we suggested that strings could be allowed in explicit This-bindings. That solution, however, is implementation dependant and rather ad-hoc.

Alternatively, we could increase JL's expressiveness by giving programmers greater control over how instantiated types are named. For instance, the ex-

tended syntax shown below could be used to explicitly designate instantiation names:

```
M<"MyName", Z<>>  // Example from previous section.
```

The above code creates the class named `M_MyName` as the leaf class in the mixin-generated hierarchy. Knowing the name of the leaf class allows us to bind **This** in all generated classes without using the name-token algorithm. Thus, circular dependencies are avoided and the code that we were unable to express in the last section we can now express:

```
Z<:M_MyName> z = new M<"MyName", Z<>>();  // OK.
```

There are other ways to give programmers control over instantiation naming. For example, a type aliasing construct like C++'s **typedef** [110] or like JL1's type equations (§6.1) could be used to specify instantiation names. The important point, however, is that in heterogeneous implementations, any language modification that affects type names needs to be carefully considered. JL currently implements an automatic name generation approach that is simple to use, though clearly imperfect. We have suggested a number of refinements that can improve JL's expressiveness, but we leave their full consideration to future work.

CHAPTER 7 EVALUATION

This chapter describes two evaluations in which we compare programming with mixins and programming using conventional techniques. The prototype applications that we build test the efficacy of mixin programming and the JL language features that support it. These evaluations show that the power of mixins can be harnessed to build applications from reusable components, and that mixin programming offers a number of benefits when compared to non-mixin object-oriented programming.

In our ACE [32,96,98] evaluation, we use mixins to reengineer a library of client/server design patterns. We compare our approach to ACE's original implementation, which uses object-oriented frameworks. We evaluate both approaches for their flexibility, usability and reusability, and we show how JL avoids problems common to frameworks. Specifically, our comparison shows that mixin programming using JL (1) avoids problems of framework evolution and overfeaturing, (2) scales better than frameworks as the number of application features increases, (3) supports a higher level of reuse than frameworks, and (4) supports application variation with more flexibility than frameworks.

In our Fidget evaluation [31], we build a software product line of graphical user interface libraries using mixin layers. We show how mixin layers increase code modularity and how this increased modularity allows libraries to be easily configured to run on platforms with widely dissimilar capabilities. We also define the Sibling design pattern, which coordinates the use of inheritance, nested

types, and the most derived types in mixin-generated hierarchies to achieve greater modularity.

To be sure, our experimental applications serve mainly as proof-of-concept exercises for the JL language; neither application has supported real users over long periods of time. Mixins need to be used in progressively more demanding environments before mixin programming can move into the mainstream; the two evaluations that we now describe are another step towards that goal.

7.1 Comparing JL and OO Frameworks

As mentioned in the Introduction, large software applications are difficult to develop and maintain. Object-oriented frameworks [20,59,98] represent the one of the most popular techniques used today to build large applications. Frameworks are *starter kits* that use abstract classes to provide partially implemented applications. Different applications are created from a single framework by providing different implementations of these abstract classes, so frameworks are ideal for supporting *software product lines*, which are families of related software products. Thus, frameworks are fundamentally a reuse technology.

The goal of JL is also to increase reuse. To compare JL against object oriented frameworks, we use JL to re-engineer the Adaptive Communication Environment (ACE) [96], an object oriented framework developed in C++ by Schmidt and colleagues. ACE is a well-documented, well-engineered framework that has been used in dozens of commercial and academic applications. Thus, ACE represents proven and mature framework technology and provides a standard against

which new technologies can be measured. In this evaluation, we compare application development in ACE and in JL using the following qualitative measures:

Usability – How easy is it to develop applications?

Application Flexibility – How easy is to customize applications?

Starter Kit Flexibility – How easy is it to evolve the starter kit?

§7.1.1 introduces the ACE framework. §7.1.2 describes our methodology and §7.1.3 describes the ACE and JL implementations. §7.1.4 compares the two approaches using the measures listed above. §7.1.5 concludes with a discussion of mixin programming and JL's supporting language features.

7.1.1 ACE FRAMEWORK

Schmidt and colleagues developed the Adaptive Communication Environment (ACE) [96,98] as a C++ framework for constructing client/server applications. ACE implements a core set of concurrency and distribution design patterns that provides an infrastructure for building customized applications. In general, C++ applications built using ACE require less effort to develop and exhibit greater flexibility, reliability and portability than C++ applications built using ad-hoc methods.

ACE is implemented in three broad layers [116]. The *System Adaptation* layer provides operating system portability. The *System Services* layer provides an object-oriented interface to the Adaptation layer. The *Distributed Design Patterns* layer implements collaborations useful in distributed applications. We

briefly describe some of the services and design patterns essential to building client/server applications using ACE.

7.1.1.1 System Services

ACE provides a Timer interface and a set of concrete classes that allow applications to create, schedule, cancel, and expire timers. Timers can be reoccurring and can be stored in specialized data structures for efficient access. ACE also provides Message Queues modeled after those found in UNIX System V [108].

7.1.1.2 Task

The ACE Task (Figure 67) is a design pattern for asynchronous processing. In its simplest form, an ACE Task is an object-oriented encapsulation of zero or more threads that perform application-specific work. A Task also contains a Message Queue to store client requests for later processing by the Task's worker threads.

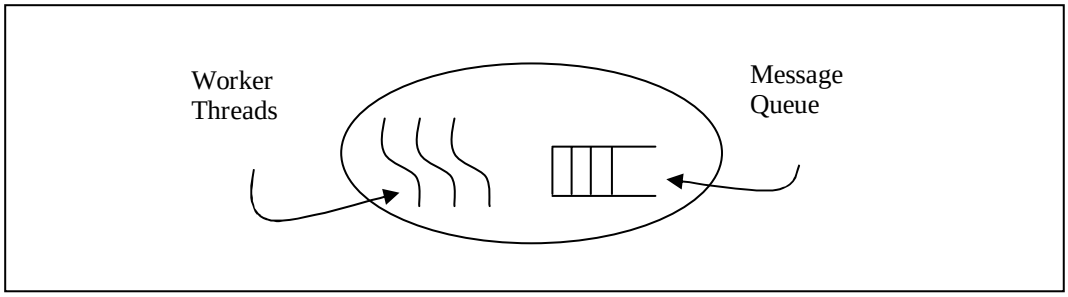


Figure 67 - ACE Task Object

The Task interface includes methods to initialize, activate and terminate a Task. Worker threads execute a virtual call-back method whose implementation

is supplied by the user through subclassing. Tasks communicate by queuing requests on each other's Message Queues.

7.1.1.3 Reactor

The ACE Reactor [99] implements a design pattern for concurrent event dispatching among multiple clients. Clients, who implement the Event Handler interface, register interest in particular events monitored by the Reactor. When an event occurs, the Reactor issues a callback to the appropriate method in registered client objects. Figure 68 shows that Reactors can monitor multiple event sources, including timers, I/O ports, operating system signals, and application level notifications.

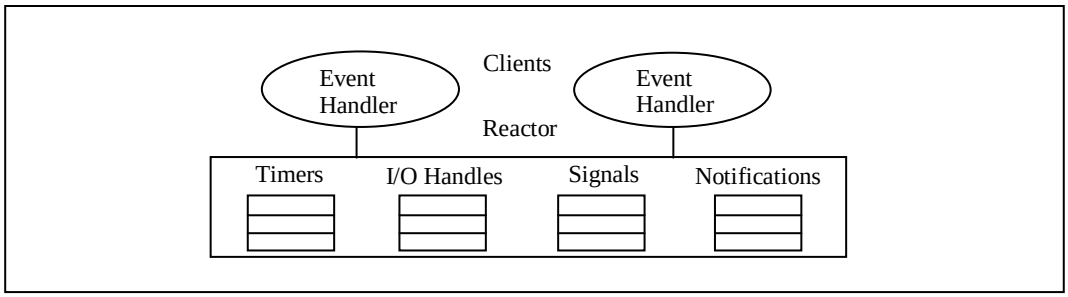


Figure 68 - ACE Reactor and Client Objects

The Reactor interface supports static methods that provide access to a default Reactor instance, as well as methods to create and manage multiple Reactors. Other methods allow clients to register, cancel, suspend and resume interest in events of all types.

7.1.1.4 Acceptor/Connector

The ACE Acceptor/Connector [97] design pattern decouples session establishment and initialization from application processing in a distributed environment. The pattern also abstracts the underlying transport stream so that different types of streams, such as TCP, Unix sockets, and pipes, can be substituted for one another. Acceptors and Connectors are factory classes [44] that come in complementary pairs: Acceptors handle the passive side of session initiation and Connectors handle the active side. These factory classes orchestrate a session initiation protocol by creating and invoking the other classes that participate in the collaboration.

Collaborators in the Acceptor subpattern are the Acceptor factory itself, a concrete stream-acceptor, a Service Handler, and a Reactor. Similarly, collaborators in the Connector subpattern are the Connector factory, a concrete stream-connector, a Service Handler, and a Reactor. Service Handlers are ACE Tasks that implement the Event Handler interface and have a stream field. Concrete acceptors and connectors provide passive and active session initiation for specific types of transport streams.

The three-phase Acceptor protocol is illustrated in Figure 69. Each Reactor notification is preceded by an appropriate event registration (not shown). The Acceptor factory directs the first two phases of the protocol, the connection initialization and service initialization phases. The Acceptor has no role in the third phase in which the Service Handler communicates independently with its peer, using the Reactor as needed. The three-phase Connector protocol is defined simi-

larly. Both protocols can be customized by overriding methods that implement each phase.

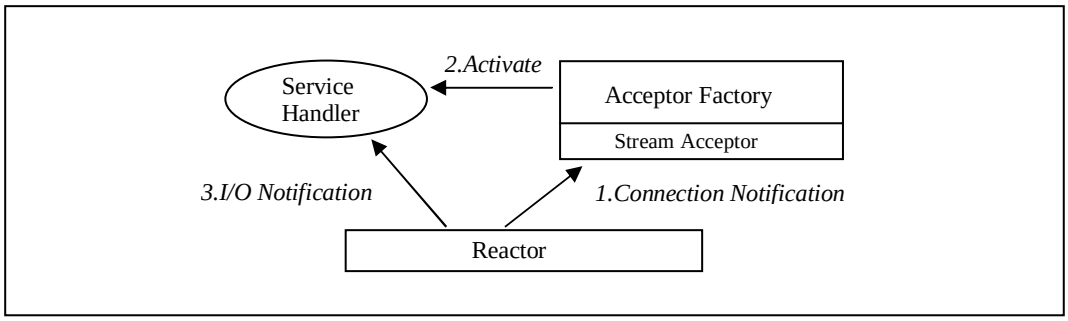


Figure 69 - Acceptor Collaboration

7.1.2 METHODOLOGY

Both JL and frameworks rely on interfaces defined during domain analysis to guide the development process. Both approaches provide starter kits of partially assembled applications, but they differ in the way in which applications are created. Frameworks provide partially assembled applications that use interfaces to define *variation points*; programmers then create applications by supplying concrete classes at all variation points. JL uses interfaces to define groups of interchangeable components that programmers then compose to build complete applications. Our goal is to compare these two approaches using the three measures described above in §7.1: usability, application flexibility, and starter kit flexibility.

To compare JL against frameworks, we used JL to re-engineer a ***subset*** of ACE that captures the sophistication of the original. Thus, we implemented the primary design patterns found in ACE necessary for building ACE-style client-

server applications, but we typically did not implement all of the features in an ACE class. The result is a few thousand lines of JL code that delivers a deep slice through ACE's layered architecture, from the application interface down to the network protocols. While our system does not come close to replicating all the function of ACE's 125K lines of code, missing functionality can be added by writing additional layers that are conceptually identical to those we have already written.

For the purpose of comparing development techniques, a complete and exact replication of ACE is not necessary. For example, our implementation uses the standard Java sockets library, which does not support a multiple port I/O call like Unix *select()* [108]. We simulate this capability by using a thread for each port, which is clearly undesirable in real-world applications, but sufficient for studying the structure of JL applications built using ACE design patterns.

We also ignore differences between JL and ACE that stem from disparities between Java and C++. For instance, many ACE classes explicitly declare synchronization parameters and methods to manage concurrency. In JL, this function is largely handled by Java's built-in multithreading support. Similarly, small differences in function, such as support for tracing and inspection during debugging, are also factored out of the comparison.

All of the services and design patterns described in §7.1.1 have been implemented in JL. Throughout our discussion, all ACE C++ classes are prefixed with "ACE_." JL classes and interfaces have unprefixed names, though all JL interfaces carry the "Ifc" suffix.

7.1.3 ACE AND JL IMPLEMENTATIONS

To provide a concrete basis for comparing JL and ACE, we now discuss the details of the two implementations. We focus on the Timer and Task design patterns, which are representative of how all ACE patterns are implemented in JL: We start with an ACE interface, decompose it into several smaller JL interfaces, and then implement these interfaces in single-feature JL layers. ACE code is described, but not shown, due to its conventional nature.

7.1.3.1 Timer

In ACE, the C++ class `ACE_Timer_Queue_T` defines the complete Timer public interface. The interface includes methods to schedule, cancel and expire timers; to retrieve and remove the next timer; to calculate the time until the next timer pop; to manage time skew; and to set the time-of-day source. Protected methods are also defined. Classes that implement this interface support all methods.

By contrast, the base JL timer interface, `TimerIfc`, declares only four `schedule()` methods. Figure 70 shows the structure of the basic JL timer class, `TimerExtensible`, which implements this interface and takes two type parameters. The first type parameter requires a subclass of `TimerAbstract` that implements the `TimerIfc` (not shown). This type parameter is mixed-in as the superclass. The second type parameter implements the `TimerSortedMapIfc` interface, which provides a container for timer objects. `Timer1` illustrates a simple use of `TimerExtensible` appropriate for applications that only schedule timers.

```
class TimerExtensible<T extends TimerAbstract implements TimerIfc,
                      U implements TimerSortedMapIfc> extends T {…}

class Timer1 extends TimerExtensible<TimerAbstract, TimerTreeMap> {}
```

Figure 70 - Simple JL Timer

In JL, advanced timer features are encapsulated in their own parameterized classes for easy composition. Figure 71 shows the `TimerCancelByTime` class that supports timer cancellation. This class inherits from its type parameter, `T`, which is constrained to implement `TimerIfc`. All instantiations of `TimerCancelByTime` implement interfaces `TimerIfc` and `TimerCancelByTimeIfc`. Features that support query, expiration and other optional operations are defined in a similar way using mixins and constrained type parameters. `Timer2` illustrates a timer that supports both cancellation and query (not shown).

```
class TimerCancelByTime<T extends TimerAbstract implements TimerIfc>
  extends T implements TimerCancelByTimeIfc {…}

class Timer2 extends
  TimerCancelByTime<
    TimerExtensible<TimerAbstract, TimerQueryId<TimerTreeMap>> > {}
```

Figure 71 - Complex JL Timer

7.1.3.2 Task

In ACE, the C++ template class `ACE_Task` defines the complete Task public interface. The interface includes public methods to activate and manage

threads; to initialize, read, write and manage a Message Queue; and to manage Tasks in the context of a Module. ACE Modules are bi-directional message streams made up of pairs of Tasks.

The JL Task interface is defined in `TaskIfc` and declares only thread activation methods. As with Timers, auxiliary interfaces are defined to support optional features. For example, the `TaskQueueIfc` interface supports Message Queue operations and the `TaskInterruptIfc` interface supports the interruption of threads. Again, features are mixed and matched to customize Tasks as needed.

7.1.3.3 Interfaces

To understand the differences between JL and ACE, it is crucial to understand how interfaces are used in the two approaches. JL's `TimerIfc` interface is *narrow* because it contains four methods and supports only the most rudimentary features used by almost all applications that require timers. Other narrow interfaces are used to declare optional features whose implementations can be composed.

By contrast, ACE Timers use a one-size-fits-all approach and implement all possible features in every Timer class. Thus, the *wide* `ACE_Timer_Queue_T` interface supports a large number of features, many of which are not needed in most applications. For example, the interface declares 20 methods, some exposing functors and iterators that are not commonly used. In the §7.1.5, we argue that wide interfaces do not stem from poor design, but rather represent an unavoidable technology-based tradeoff.

To summarize, ACE uses a small number of wide interfaces, while JL uses a larger number of narrow interfaces. For each ACE interface used in our evaluation, Table 5 shows the number of declared methods, the number of narrow JL interfaces produced, and the average number of methods in the JL interfaces.[20]

	Timer	Queue	Task	Reactor	Acceptor	Connector
ACE Width	20	24	15	66	5	5
No. of JL Interfaces	13	13	10	27	3	4
Average JL Width	1.5	1.8	1.5	2.4	1.7	1.3

Table 5 - Interface Width in ACE and JL

7.1.4 COMPARISON

In this section, we compare ACE and JL using the three measures described in §7.1.

7.1.4.1 Usability

How easy and effective is software development using the two approaches? We answer this question by comparing interface usage in JL and ACE.

ACE's wide interfaces are more complex and therefore harder to use than JL's narrow interfaces. Wide interfaces not only require users to learn more methods, but the methods themselves sometimes take more parameters. For example, the `ACE_Task` constructor takes a Message Queue parameter, thereby forcing all Task users to understand something about queuing. In JL, the Message

[20] Factoring out differences between C++ and Java.

Queue type does not appear in Tasks that do not implement the Message Queue feature.

The use of narrow, less complex interfaces in JL also leads to smaller executables. We saw how JL Timer classes could easily be constructed with the exact set of features required by an application and no more. ACE Timers, on the other hand, have uniformly large executables because of the width of the interface that they must support.

JL's narrow interfaces can also lead to lower execution overhead. For example, JL Tasks that don't implement `TaskQueueIfc` avoid the overhead of allocating and initializing a Message Queue, costs incurred by every ACE Task.

JL's ability to precisely customize code to its application environment leads to simpler interfaces and smaller, faster implementations. All these characteristics increase the likelihood that JL code will meet the needs of application programmers and, as a consequence, be used.

In terms of maintenance, there is a tradeoff between the number and size of interfaces. An excessive number of small interfaces in JL could be just as unmanageable as excessively large interfaces in frameworks. In our evaluation, however, we found that reasonable interface design avoids the worst-case management problems in both JL and ACE.

Finally, while frameworks apparently give programmers more functionality by providing partially assembled applications, JL can do the same by delivering predefined or canned layer compositions. These canned compositions can even be packaged as frameworks.

7.1.4.2 Application Flexibility

To what extent do ACE and JL allow applications to be constructed with precisely the desired set of features?

The use of wide interfaces in ACE means that any implementation of a service, such as the Timer service, must support all possible methods. In addition, applications that use these services do not have the ability to pick and choose optional features, though new optimization techniques may remove unused code from the application after the fact [121].

On the other hand, the use of narrow interfaces in JL allows each optional feature to be implemented in its own class. These optional features can then be composed to yield a great variety of customized types for use in applications. Table 5 on page 219, for example, shows that any of 27 separately implemented Reactor features can be used to generate a Reactor. This yields 2^{27} possible feature combinations, even if we assume no duplicates and a total ordering among features. In JL, we compose optional features on demand rather than in advance, allowing JL to avoid the feature combinatorics problem described in §2.1.

7.1.4.3 Starter Kit Flexibility

This section compares the ability of JL and frameworks to support changes to their starter kits. We first consider how the two approaches support evolving client needs. We then discuss the more specific issue of adding features to the starter kit.

7.1.4.3.1 Evolving Client Needs

A well-designed framework strikes a balance between what to include in the framework and what to exclude. The framework will ideally include all code that is common across many applications. If the framework includes too many features, the interface becomes overly complex and the framework becomes less usable. If the framework omits commonly needed code, multiple applications will have to implement the missing features independently. These problems are commonly referred to as *overfeaturing* and *code replication*, respectively [34].

As well designed as ACE is, it still exhibits overfeaturing and code replication. For instance, `ACE_Reactor` includes methods that support the singleton design pattern [44], which is useful in applications that require only one Reactor, but which is confusing in applications that use multiple Reactors. Thus, what is appropriate for one application may appear to be overfeaturing to another. On the other hand, ACE does not support authentication, authorization or data privacy. Unless the ACE framework is updated, each application requiring security must independently develop its own network security solution outside of the framework.

The problems of overfeaturing and code replication are rooted in the fundamental and somewhat rigid distinction that all frameworks make between framework code and application code [10]. Deciding what to include in a framework is always a compromise based on domain knowledge and the requirements of future users, both of which are likely to change over time.

On the other hand, JL promotes code reuse with its ability to selectively mix and match features. JL classes are grouped according to the interfaces they implement. Adding a new capability to a set of starter kit classes usually has minimal impact because of the loose coupling between classes and the orthogonal nature of feature implementations. Adding new starter kit classes is no different than adding application classes.

7.1.4.3.2 Adding Features to the Starter Kit

Suppose a framework needs a new feature that requires changes to its core classes. One approach is to modify existing framework classes while maintaining backward compatibility as much as possible. This approach is not feasible if currently supported applications are intolerant of changes in their binary representation. Applications that store objects persistently or that are conservatively managed for safety reasons often fall into this category. This need to maintain compatibility between separately evolving framework and application code is known as the *framework evolution* problem [34].

An alternate approach is to implement the new feature in new framework classes. Unfortunately, this approach spawns a new class hierarchy that is parallel to the existing one, creating a potentially large amount of nearly identical new code to maintain. Figure 72 illustrates how a new subtree is created when changes for class `B` are instead implemented in a new class named `b`. Class `b` is a subclass or a copy of class `B`. If child `C` of `B` needs to support the new feature, it does so through its proxy class, `c`, in the new subtree.

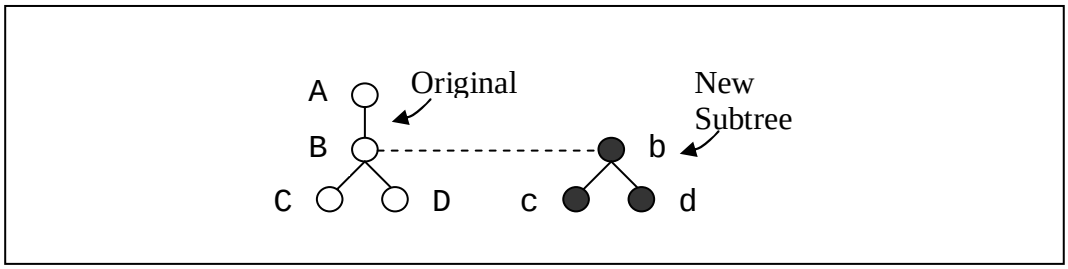


Figure 72 - Framework Evolution

In JL, evolution can be implemented using the same two approaches available to frameworks. If changing an existing class is not desirable, a new class can be created, typically using inheritance, to incorporate the changes. The loose coupling of JL classes, however, means that the original class is typically not part of a predefined hierarchy, so no parallel subtree is spawned. There is no compatibility problem because applications can be generated using either the new or old classes.

7.1.4.3.3 Changes in Domain Analysis

If new features require the refactoring of important interfaces, then JL and frameworks are equally susceptible to disruption because they both rely on good domain analysis to define interfaces appropriately.

7.1.5 DISCUSSION

Mixins are the key to JL's power and flexibility because they allow code to be varied in a new way. In addition to the techniques that support code variation in ACE—subclassing, type parameters, and runtime initialization parameters—JL allows a class's supertype to be varied using mixins. In previous work

[10], we proposed that frameworks themselves could be implemented more flexibly using a layered component technology.

Mixins also allow features to be mixed and matched so that new types can be built in a stepwise manner. In JL, we widen interfaces to support the exact feature set that an application requires by encapsulating features in their own classes and composing them. JL uses mixins to solve the feature combinatorics problem without resorting to wider than necessary interfaces. In JL, unused feature combinations are never materialized.

Mixins work because they defer the specification of parent/child relationships from definition time to composition time. This late binding promotes JL's stepwise refinement model that in turn encourages interfaces to be smaller, less complex, and feature-specific. ACE, and frameworks in general, use non-parameterized inheritance to lock in parent/child relationships and create application skeletons. This rigidity forces the use of wide interfaces to avoid the combinatorial explosion in the number of classes that would result from materializing all feature combinations in advance (§2.1).

In our reengineering effort, we used a number of JL's new language features to support mixin programming. For example, we used constructor propagation (§4.4) to automatically generate constructors for different Task types. Whenever the `TaskQueue` mixin, which implements `TaskQueueIfc` (§7.1.3.2), is used in a composition, Task constructors are automatically augmented with a Message Queue parameter.

We also used the **This** implicit type parameter (§4.3) in our evaluation. Figure 73 shows how **This** is used to implement the singleton Reactor feature in the `ReactorSingle` mixin. Singleton Reactors are useful in applications that require only one Reactor instance. The code uses **This** to create and store an instance of the most derived class in the mixin-generated hierarchy.

```
class ReactorSingle<T implements ReactorIfc> extends T
 {private static This _inst;
  public static This instance(){
   if (_inst == null) _inst = new This();
   return _inst;} }
```

Figure 73 - The Singleton Reactor Feature

To gauge the expressiveness of JL's semantic checking constraints (§4.5), we applied semantic constraints to our reengineered ACE components. These constraints could only be applied on paper since the semantic checking facility is not currently implemented in JL. We found that the **requires unique** constraint was most often used in our code. For example, we augmented the definition of `TimerCancelByTime` (Figure 71 on page 217) with **requires unique** because only one version of the timer cancellation code is ever needed in a composition. In our ACE evaluation, we were able to express all the constraints we needed, but these results can only be considered preliminary.

This concludes our discussion of the ACE evaluation; we now describe the Fidget evaluation.

7.2 Applying Mixin Layers in Fidget

For many years, software portability meant running software on different general-purpose computers, each with its own operating system and architecture. Software developers minimized the cost of supporting multiple platforms by reusing the same code, design, and programming tools wherever possible. Today, miniaturization has led to a wide diversity of computing devices, including embedded systems, cell phones, PDAs, set-top boxes, consumer appliances, and PCs. Though these devices are dissimilar in hardware configuration, purpose and capability, the same economic forces that necessitated software reuse among general-purpose computers are now encouraging reuse across these different classes of devices.

To make it easier to reuse code across devices, a number of standardization efforts are defining new Java runtime environments [55]. These environments are customized for various classes of devices while they still remain as compatible as possible with the Java language, JVM, and existing libraries. For example, Sun's KVM [111,113] virtual machine, which is designed to run on devices with as little as 128K of memory, has removed a number of Java language features, including floating point numbers and class finalization, and a number of JVM features, such as native methods and reflection. In addition, the runtime libraries and their capabilities have also been reduced to accommodate limited memory devices. This redesign of the Java libraries leads to two questions that directly concern code reuse:

- How do we scale an API to accommodate different device capabilities?
- How do we reuse the same library code across different devices?

This section explores the above questions by designing and implementing a graphical user interface (GUI) that works on cell phones, Palm OS™ devices [90], and PCs. The challenge is to provide a single GUI code-base that runs on all these devices yet accommodates the input, output, and processing capabilities of each device. For example, a device may or may not support a color display, so in building our libraries we would like to be able to easily include or exclude color support.

Our solution is to use Java Layers to encapsulate features that crosscut multiple classes, such as support for color, to a degree that is not possible using standard programming technologies. Our evaluation tests the hypothesis that mixins and mixin layers provide a convenient mechanism for encapsulating cross-cutting concerns. We test this hypothesis by building *Fidget*, a flexible widget library, and by showing that its design and implementation are effective: We show that Fidget can be easily customized for various execution environments and that Fidget libraries are easy to use. The main contributions of our Fidget evaluation are as follows:

1. We demonstrate how mixins, supplemented by JL's supporting language features, are effective in customizing software for disparate platforms.

2. We define the Sibling design pattern and demonstrate how it can increase code modularity.
3. We add to the growing body of evidence that mixins, mixin layers, and the programming model of layered refinement (§2.3.1) are effective in increasing code reuse.

§7.2.1 describes Fidget's design goals and §7.2.2 discusses the methodology we use to evaluate our design approach. §7.2.3 presents Fidget's design and §7.2.4 shows how Fidget components are used. §7.2.5 discusses the issues raised in our evaluation.

7.2.1 DESIGN GOAL

The goal of building graphics libraries that accommodate dissimilar devices is to demonstrate that we can mix and match *features* depending on the target execution environment, where a feature is some GUI characteristic such as color support. This goal of flexible feature selection highlights two requirements of reusable code: (1) modularity and (2) easy composition. Specifically, the code for a feature should be completely encapsulated in a module, and these modules should be easy to compose with one another.

In §2.2, we described how current technology limits our ability to reuse code and to build software from components. In this evaluation, we use JL to build Fidget incrementally in layers (§2.3.1) and then evaluate our design approach.

7.2.2 METHODOLOGY

In this section, we describe the methodology that we use to evaluate our design approach. We describe Fidget libraries, how they are used, and the limitations of our implementation.

We use mixins in JL to design and implement a number of graphics library features. We then compose these features to generate specialized instances of Fidget libraries for various devices. The generated graphics libraries are not complete GUIs, but are prototypes used to validate our design approach. So, for example, we provide some basic look-and-feel options and describe how a complete platform-specific skin would be implemented using our design, but we do not provide the complete implementation.

We demonstrate that Fidget libraries can be easily configured for cell phones, Palm OS devices, and PCs. We use Fidget libraries to implement simple applications, and we compare application development using Fidget against the use of a more conventional GUI library. Since our goal is to evaluate the usefulness of Java Layers for library and application development, we do not write low-level graphics code to interface directly with each device's operating system. Instead, we scaffold our code on top of a small subset of the Java graphics library present on each device.

Our target PC environment is standard edition Java 1.3.1 and its development kit (SDK) [114]. We use the Java 2 Micro Edition (J2ME) Wireless Toolkit 1.0.3 Beta [55] for our cell phone and Palm environments. Our Palm OS tests are run on the Palm OS Emulator version 3.2 [90].

7.2.3 FIDGET DESIGN

In this section, we describe how the Sibling design pattern forms the basis of Fidget's design. We also describe how we use constructor propagation to avoid defining constructors by hand. To provide context for this discussion, we first discuss Fidget's architecture, the way it uses mixin layers, and its component design.

7.2.3.1 Architecture

In this section, we describe Fidget's architecture. Fidget is structured as a stack of the three architectural layers highlighted in Figure 74: the hardware abstraction layer, the kernel layer, and the user layer. On the bottom, the hardware abstraction layer (HAL) interacts with the underlying device's graphics system and is the only Fidget code that is device dependent. On top, the user layer is a thin veneer that provides a familiar, non-nested, class interface to application programmers. Our discussion focuses on the kernel layer in the middle.

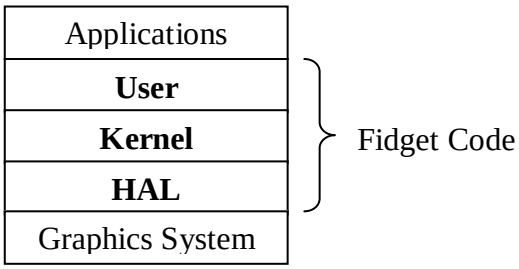


Figure 74 - Fidget's Architectural Layers

The kernel layer defines all widgets and all optional widget features. The kernel sits on top of the HAL and uses the HAL's drawing and event handling capabilities to create displayable widgets. Fidget widgets are modeled after those of Java's AWT [45,114], so widget classes such as Window, Button and TextField serve the same purpose in Fidget as their analogs do in AWT. The kernel implements nine such widgets, which is sufficient for our prototyping purposes. There is only one kernel code-base, even though some optional features cannot be used with all devices.

The Fidget kernel uses a *lightweight* implementation [45] to accommodate devices with constrained memory resources. Lightweight widgets do not have associated peer widgets in the underlying graphics system, which for Fidget is either the SDK or J2ME. Thus, a Fidget window that displays two buttons and a text field creates only one widget, a window, in the underlying Java system. Fidget then draws its own buttons and text field on this underlying window.

7.2.3.2 Using Mixin Layers

In this section, we describe how we use mixin layers (§2.3.3) to build kernel layer widgets. The code below shows simplified versions of the base Fidget class and the mixin layer that adds color support:

```
class BaseFidget<> {
       public class Button {…}
       public class CheckBox {…} …}

class ColorFidget<T> extends T {
       public class Button extends T.Button {…}
       public class CheckBox extends T.CheckBox {…} …}
```

`BaseFidget` takes no explicit type parameters and contains all of the widget classes; we show two widget classes, `Button` and `CheckBox`, in the code

above. The `ColorFidget` mixin layer extends the behavior of each nested class in `BaseFidget` with color display support. In this way, feature code scattered across multiple classes is encapsulated in a single mixin layer.

We now describe how we use deep conformance (§4.2) to structure Fidget's mixin code. Normally, a Java class that implements an interface is not required to implement the interface's nested interfaces; deep conformance extends Java's interface constraints to include nested interfaces. In Figure 75, we use JL's **deeply** modifier to enforce the condition that for each nested interface in `FidgetTkIfc`, the (revised) `BaseFidget` must define a public nested class with the same name, and that nested class must implement the corresponding interface. Thus, `BaseFidget.Button` implements `FidgetTkIfc.Button`.

```
interface FidgetTkIfc {
  interface Button {…}
  interface CheckBox {…} …}

class BaseFidget<>
 implements FidgetTkIfc deeply {
  public class Button implements FidgetTkIfc.Button {…}
  public class CheckBox implements FidgetTkIfc.CheckBox {…} …}

class ColorFidget<T implements FidgetTkIfc deeply>
 extends T deeply {
  public class Button extends T.Button {…}
  public class CheckBox extends T.CheckBox {…} …}
```

Figure 75 - Deep Conformance in Fidget

In addition, when **deeply** is used in a mixin class's extends clause, the superclass's public structure is preserved in the instantiated subclass. This means that if a class nested in a mixin has the same name as a public class nested in the mixin's superclass, then the mixin's nested class inherits from the superclass's

nested class. In the `ColorFidget` mixin in Figure 75, `Button` and `CheckBox` must subclass their respective superclass members because (1) `ColorFidget` deeply extends its superclass and, (2) any actual superclass must contain public `Button` and `CheckBox` classes due to the constraint on type parameter `T`.

7.2.3.3 Reusable Components

In the previous sections, we described Fidget's architecture and how mixin layers are used to build Fidget's kernel code. In this section, we discuss Fidget's kernel classes in greater detail.

Fidget's kernel classes provide the foundation and optional features for all Fidget GUIs. Fidget design is based on the above-described `BaseFidget` class, which provides the minimal implementation for each widget in a nested class. The nested widget classes are `Button`, `CheckBox`, `CheckBoxGroup`, `Label`, `Panel`, `TextArea`, `TextComponent`, `TextField`, and `Window`.

Optional features are implemented in mixin layers that deeply conform to `BaseFidget`. These mixin layers can contain code for one widget class, or they can implement crosscutting features and contain code for more than one widget class. For example, the `TextFieldSetLabel` layer affects only one class by adding the `setLabel()` method to `TextField`. Conversely, the `Light-WeightFidget` layer implements lightweight widget support and contains code for most widgets. Fidget's mixin layers and the features they implement are listed below.

Non-Crosscutting Kernel Mixins

`ButtonSetLabel`	– Re-settable Button label
`BorderFidget`	– Draws container borders
`CheckboxSetLabel`	– Re-settable Checkbox label
`TextComponentSetFont`	– Changeable fonts
`TextFieldSetLabel`	– Re-settable TextField Label

Crosscutting Kernel Mixins

`AltLook`	– Alternative look and feel
`ColorFidget`	– Color display support
`EventBase`	– Basic event listeners
`EventFidget`	– All event listeners/handlers
`EventFocus`	– Focus event handling
`EventKey`	– Key event handling
`EventMouse`	– Mouse event handling
`LightWeightFidget`	– Lightweight support

`BaseFidget` also contains two nested classes that serve as superclasses for the nested widget classes. `Component` implements common widget function and is a superclass of all widgets. `Container`, a subclass of `Component`, allows widgets to contain other widgets. `Window` is an example of a container widget. Defining these superclasses in `BaseFidget` has important design consequences, which we now explore.

7.2.3.4 The Sibling Pattern

To enhance code modularity, the Sibling design pattern uses inheritance relationships between classes that are nested in the same class. The pattern itself can be implemented in Java, but mixin layers make it more convenient to use. We begin our discussion of this pattern by looking at a problem that occurs when certain crosscutting features are implemented with mixin layers. We then show

how the Sibling pattern solves this problem and how JL language support simplifies the solution.

The advantage of nesting `Component`, `Container` and all widget classes inside of `BaseFidget` is that a single mixin layer can affect all these classes. We re-introduce `BaseFidget` in Figure 76, this time showing the widget `Button` and its superclass `Component`. In Fidget, features like support for color modify the behavior of `Component` as well as its widget subclasses.

```
class BaseFidget<>
 implements FidgetTkIfc deeply {
  public abstract class Component {
   implements FidgetTkIfc.Component {…}
  public class Button
   extends Component
   implements FidgetTkIfc.Button {…}

class ColorFidget
 <T implements FidgetTkIfc deeply>
 extends T deeply {
  public class Component
   extends T.Component {…}
  public class Button
   extends T.Button {…} …}

ColorFidget<LightWeightFidget<BaseFidget>>
```

Figure 76 - Incorrect BaseFidget

There is, however, a potential pitfall when parent and child classes are nested in the same class. To see the problem, Figure 76 also depicts the `ColorFidget` mixin and an instantiation of a Fidget GUI with color support. The instantiation includes the `LightWeightFidget` mixin (code not shown), which is structured the same as `ColorFidget`.

The class hierarchies generated by the instantiation are shown in Figure 77. The enclosing classes form a class hierarchy, as do like-named nested classes. In addition, `Button` inherits from `Component` in `BaseFidget`. Notice that `ColorFidget.Button` does not inherit from `ColorFidget.Component`, which means that the color support in the latter class is never used. As a matter of fact, it would be useless for any mixin layer to extend `Component` because no widget will ever inherit from it.

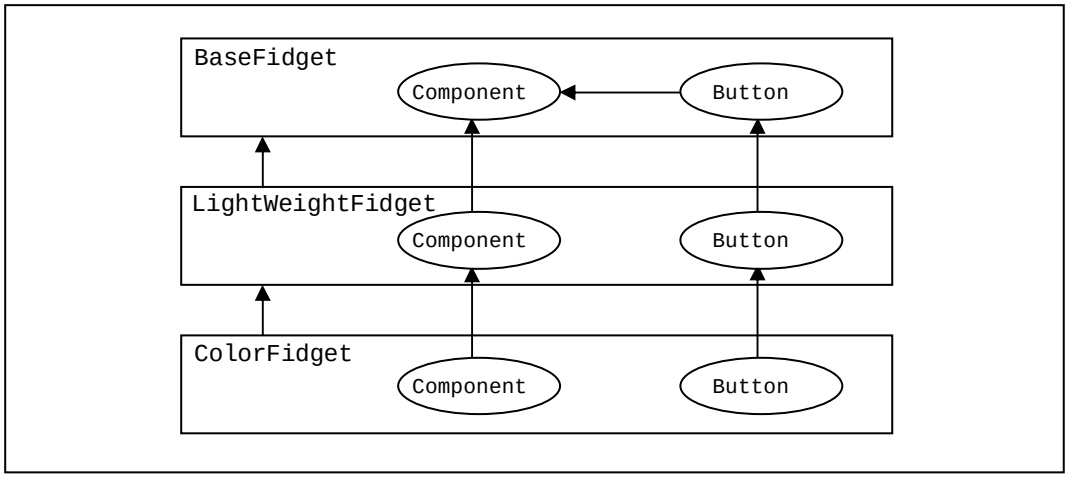


Figure 77 - Incorrect Hierarchy

The inheritance relationship we really want is shown in Figure 78, where `ColorFidget.Button` inherits from all the `Button` classes and from all the `Component` classes in the mixin-generated hierarchy. We call this the *Sibling pattern*, which we define as the inheritance pattern in which a nested class inherits from the *most specialized subclass* of one of its nested siblings. In Figure 78, `BaseFidget.Button` inherits from the most specialized subclass (`Color-Fidget.Component`) of its sibling (`BaseFidget.Component`).

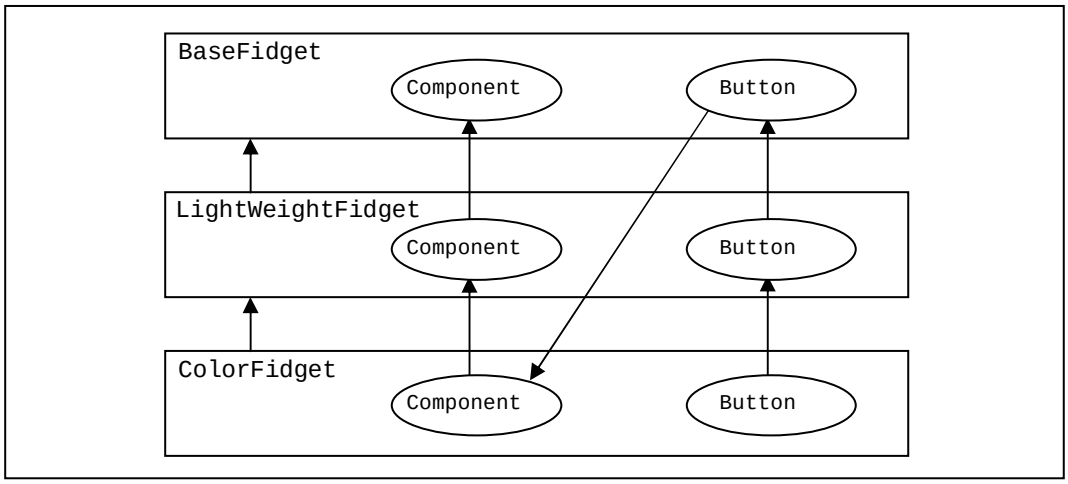


Figure 78 - Sibling Pattern Hierarchy

The Sibling pattern can be implemented in Java by using a distinguished name for the leaf class of all mixin-generated hierarchies. Once this well-known, predetermined name is established by programming convention, it can be used in any class or mixin in the application. This solution, however, limits flexibility and can lead to name conflicts when different instantiations are specified in the same package.

JL provides a better way to express the Sibling pattern using its implicit **This** type parameter (§4.3). Figure 79 shows how `BaseFidget`, which declares no type parameters explicitly, uses its implicit **This** parameter to implement the Sibling pattern. JL binds **This** to the leaf class in the generated hierarchy, which in our example is `ColorFidget`. The redefined `Button` class below now inherits from `ColorFidget.Component`.

```
class BaseFidget<>
 implements FidgetTkIfc deeply {
  public abstract class Component
   implements FidgetTkIfc.Component {…}
  public class Button
   extends This.Component
   implements FidgetTkIfc.Button {…} …}
```

Figure 79 - Correct BaseFidget

The Sibling pattern allows a Fidget layer to extend individual widget classes and their common superclass simultaneously. In this way, established object-oriented methods of class decomposition, in which common function is placed in superclasses, are extended to work with mixins layers. In Fidget's mixin layers, refinements to `Component` are inherited by all widget classes in all layers. This brings us to the last topic in our design discussion, the use of constructors with stepwise refinement.

7.2.3.5 Constructor Propagation

In Fidget, we use constructor propagation (§4.4) to automatically generate constructors in mixin layers. One measure of the effectiveness of constructor propagation is the number of constructors that do not need to be hand-coded. In our design, all constructors available in `BaseFidget` also need to be available in the most derived classes in mixin-generated hierarchies. `BaseFidget` defines twenty constructors with the **propagate** modifier. On average, the thirteen kernel layers that extend `BaseFidget` declare just over one constructor each, which in-

dicates that constructor propagation largely relieves programmers from hand replicating constructors in mixin layers.

7.2.4 USING FIDGET

In the section, we describe how to generate and use customized Fidget libraries. We first look at how custom Fidget libraries are specified. We then discuss how applications use a Fidget library in place of Java's AWT library. Finally, we give details about the Converter application, which uses Fidget libraries on three different platforms.

7.2.4.1 Building Fidget Libraries

To build a Fidget library, we first select the SDK or J2ME hardware abstraction layer based on the target device's Java support. This layer, which corresponds to the HAL in Figure 74 on page 231, provides a small set of line and curve drawing primitives that is consistent across all platforms.

Next, we specify and compile the features that we need in our library. The code implementing the different features resides in mixin layers in the kernel package, which corresponds to the kernel layer in Figure 74. The actual Fidget libraries are assembled in the user layer, which we implement in the in the `widget` package. The code below shows the feature selection for two different libraries.

```
package widget;
import kernel.*;

class Fidget extends
      AltLook<EventFidget<LightWeightFidget<BaseFidget<>>>> {}

class Fidget extends
      ColorFidget<ButtonSetLabel<EventKey<EventMouse<
         EventBase<LightWeightFidget<BaseFidget<>>>>>>> {}
```

Both of the above libraries are lightweight implementations, the only kind currently available in Fidget. The first library supports all events and, by overriding the drawing methods in `LightWeightFidget`, provides an alternative look and feel. The second library supports color displays, re-settable labels, and key and mouse event handling. If a library feature is not supported by the device it runs on, then executing the feature code either has no effect or throws an exception.

In addition to the `Fidget` class, the user layer contains wrapper classes for each widget. These classes allow Fidget widgets to replace AWT widgets in application code. Below we show the definitions for the `Button` and `Window` wrapper classes.

```
public class Button extends Fidget.Button {}
public class Window extends Fidget.Window {}
```

To use a Fidget library, application code simply imports `widget.*` and uses the Fidget widgets in the same way that AWT widgets are used. The following sample code functions in a similar way using either Fidget or AWT. The code creates a window with a single button. The button's label is set to "ButtonLabel" and then the window is displayed on the screen.

```
// import widget.* or java.awt.*
public class Sample {
  public static void main(String[] args) {
    Window win = new Window(…);
    Button b = new Button(“ButtonLabel”);
    win.add(b);
    win.setVisible(true)
  } }
```

7.2.4.2 The Converter Application

As part of the evaluation of Fidget, a simple application named Converter was built on three target devices: JDK 1.3.1 on Linux, J2ME on Palm OS, and J2ME on a cell phone emulator. Each device has its own version of Converter, which converts between metric and US lengths.

In all versions, the `Converter` class drives the application by creating two `ConversionPanel`s, one with metric units and one with U.S. units. These two `ConversionPanel`s are added to the main window of the application, and then the window is made visible.

The Converter application code is not exactly the same across devices. This variation reflects the need for platform specific code, which adds to the porting effort. The important point, however, is that the same Fidget code-base, which is implemented in mixin layers in the `kernel` package, is used on all three platforms to control screen I/O. To understand the nature of the platform dependencies, we now describe the three versions of Converter.

Among the three versions of the Converter application, the `Converter` class varies in two ways. First, the precision of the converter is limited in J2ME environments because floating-point numbers are not available. In the JDK ver-

sion, conversion results are computed and displayed as floating point numbers. In the J2ME versions, the results are computed and displayed as integers.

The second way in which the `Converter` class varies involves application startup. In the JDK version, a `main()` method in Converter allows the application to be run from the command line. In the J2ME versions, a `J2MEConverter` class wraps the `Converter` class and implements the application interface required by J2ME.

The `ConversionPanel` class also differs across platforms. Again, the variation does not extend into the Fidget library, but is contained at the application level. In the cell phone version of the application, text fields are made smaller and certain input buttons are removed due to the physical limitations of the device. These changes are localized to the `ConversionPanel` class.

The Converter application demonstrates that (1) GUI library support can be easily configured for disparate devices using a single code-base, and (2) Fidget libraries are as easy to use as conventional GUI libraries. Once the JDK-specific version of Converter was written, porting the application to the other platforms was not difficult.

7.2.5 DISCUSSION

In this section, we discuss the rationale, advantages and alternatives for Fidget's design. We begin by describing two characteristics of mixin code that impact flexibility and usability, *layer width* and *feature granularity*.

7.2.5.1 Layer Width

When a mixin layer, or a class like `BaseFidget` that mixin layers extend, contains many nested classes, we say the layer is *wide*; otherwise, we say the layer is *narrow*. In general, wide layers have a greater ability to implement crosscutting features. However, wide layers can lead to larger, more complex classes because they can contain the code for many nested classes.

In Fidget, we define all widgets and their superclasses as sibling nested classes to increase code modularity. This organization encapsulates feature implementations that can refine any number of widgets, as the crosscutting mixin layers listed in §7.2.3 illustrate. The ability to write wide layers in Fidget, however, does not require that all layers be wide: Layers that extend a single widget only contain code for that widget. Wide layers, and the modularity they afford, allow Fidget to achieve its compositional flexibility.

In general, deciding what classes to nest in an application's layers requires careful planning. Once the decision is made, only features that crosscut the chosen nested classes can be encapsulated in a mixin layer. For example, an alternate Fidget design, which is actually the first design we tried, defines two kinds of kernel layers. The first kind is narrow and contains only the `Component` and `Container` classes. The second kind contains all the widget classes. Using this design, refinements to widgets and refinements to their superclasses would be applied separately using different sets of layers. The idea is to first select features for `Component` and `Container`, generate those classes, and then use those classes as pre-packaged superclasses for generating customized widgets.

Unfortunately, features like support for color crosscut both widgets and their superclasses. In the alternate design, color support requires that two layers, one that refines widgets and one that refines their superclasses, be used in conjunction. Fidget, however, nests all classes in the same layers, which allows us to implement color in one mixin layer.

The important design point here is that when coordinated changes need to be made to a group of classes, the classes usually should be nested in the same layers. Applications can contain mixin layers that deeply conform to different interfaces. Only those layers, however, that deeply conform to the same interface are interchangeable, and only those layers that contain all of a feature's collaborating classes can implement that feature.

7.2.5.2 Feature Granularity

The choice between fine-grained and coarse-grained layers leads to a tradeoff between incrementality and compositional complexity. In Fidget, we implemented event handling using two levels of granularity to compare each approach. Fidget supports focus, key and mouse events. The `EventBase`, `EventFocus`, `EventKey` and `EventMouse` mixins implement the fine-grained approach, which allows incremental customization based on the type of event. For devices that don't support all types of input, this approach allows more precise customization. This ability to tailor code to a platform can be used to reduce a GUI's memory footprint. On the other hand, the `EventFidget` mixin implements all event handling for all widgets, which makes adding event support a

simple matter of specifying one layer for any device. Events that never occur on a device are never handled.

The choice in mixin layer granularity is analogous to the choice in method granularity that class designers make. For mixins, just as for methods, it is sometimes desirable to support multiple granularities at once. In such situations, code replication can be avoided if the fine-grained implementation can be used to build the coarse-grained implementation. For methods, coarse-grained implementations can be built by creating new methods that bundle calls to existing fine-grained methods. For mixins, coarse-grained implementations can be built by creating new mixins from compositions of existing fine-grained mixins. These compositions would not have all their actual type parameters specified, so they represent a partial application of parameterized types. JL does not currently support such compositions.

7.2.5.3 Defining the Sibling Pattern

One of the contributions of Fidget is the recognition that the inheritance relationship between nested classes described in §7.2.3.4 is a design pattern. The Sibling pattern is noteworthy because of the way it uses nested classes, layering, and the most specialized type in a hierarchy. The pattern is useful because *in a deeply conforming mixin layer, changes to a nested class can be inherited by its sibling classes*.

The Sibling pattern's inheritance relationship has been observed in other applications [14,76], which supports the idea that the pattern should be cataloged. The intent, motivation, use and structure of the Sibling pattern have already been

described. In this section, we briefly comment on its applicability and enabling language features. A formal description of the Sibling pattern is available [25].

The Sibling pattern is most applicable when (1) nested classes are supported and (2) class hierarchies can be changed without changing class definitions. Though the pattern can be implemented in non-parametric Java, Java's fixed class hierarchies discourage the use of layers of nested classes for implementing crosscutting features, so the pattern is rarely seen. When mixin layers or similar constructs are available, the Sibling pattern allows a parent class and its children classes to be refined simultaneously. This capability makes stepwise program refinement even more powerful.

The Sibling pattern requires that the type of the leaf class in a hierarchy be available in classes that make up the hierarchy. In §7.2.3.4, we saw that though a simple naming convention is sufficient to meet this requirement, JL's **This** type parameter provides more flexibility. The Sibling pattern could also be implemented using virtual types [120].

This concludes our discussion of the Fidget evaluation; we now summarize the results of both the ACE and Fidget evaluations.

7.3 Summary

Our ACE evaluation shows that mixins provide significant advantages in terms of flexibility, usability, and reusability when compared to object-oriented frameworks. JL breaks the static binding among framework classes and instead delivers a collection of composable classes. These classes can be combined in

different ways to meet the needs of particular applications. Mixins provide the required compositional flexibility, while other language features enhance usability.

Our Fidget evaluation provides empirical evidence that an important domain like GUIs can be decomposed into feature-encapsulating components using mixin layers, and that these components can be combined into custom libraries using stepwise refinement. From a single code-base, we generated different GUI libraries for cell phones, Palm devices and PCs. These generated GUIs present conventional programming interfaces to applications and contain only the APIs appropriate for their target devices.

Both of the above evaluations show that JL's novel language support, including deep conformance, constructor propagation and the implicit **This** type parameter, increases the effectiveness of programming with mixins. This effectiveness is reflected in a number of ways, including improved constraint checking and less programmer-written code. Another measure of this effectiveness is JL's ability to easily implement the Sibling pattern, which we used in our Fidget design.

The Sibling pattern, which incorporates both type nesting and type inheritance, may at first seem complicated, but we have shown that the pattern supports a simple semantic: *In a deeply conforming mixin layer, changes to a nested class can be inherited by its sibling classes.* This semantic extends standard OO inheritance semantics to mixin layers in a useful way.

CHAPTER 8 CONCLUSIONS

This chapter summarizes our primary results and contributions; it also discusses possible directions for future research.

8.1 Results

The high cost of creating and evolving large applications drives the search for more efficient software development techniques. One way to reduce software cost is to increase the reuse of existing code. Our approach to increasing reusability is to provide language support for more powerful encapsulation and composition capabilities. In particular, we use mixins as the technological foundation for Java Layers and then show how novel language and compiler support make mixin programming more effective.

The fundamental result of our research is that *mixins become a more practical reuse technology when they are coupled with specialized language and compiler support*. Mixins provide the modularity and composition properties needed for reuse. Mixins, however, also introduce new complexity, and it is this complexity that we address in Java Layers research. We now summarize the three primary contributions of our research.

Our first contribution is to identify support needed to manage the additional complexity of mixin programming. The challenges of mixin programming have been previously documented by others; our contribution is the design and implementation of features that make mixin programming easier, more expres-

sive, and more efficient. We identified areas where mixin programming is more complicated than conventional object-oriented programming. We found that initialization is less straightforward in mixin classes than in fixed-superclass classes, so we designed and implemented constructor propagation to address initialization. We also found that mixin-generated code contains more indirection than conventional application code, so we designed the class hierarchy optimization to remove design-time layering from runtime code.

We identified areas where mixin programming does not have an analog in standard object-oriented programming, but where specialized support for mixins would increase their usefulness. We found that mixin compositions sometimes need to be restricted in ways not possible using syntactic type checking, so we designed a simple semantic checking capability. We found that explicit support for mixin layers keeps applications well-structured, so we designed and implemented deep conformance support. Lastly, we found that the most derived class in a mixin-hierarchy is distinguished because it contains all the features implemented by a mixin composition, so we designed and implemented the implicit **This** type parameter to allow symbolic references to that class.

Our second contribution is to show how mixins and their supporting features can be integrated into an existing language. We chose to extend Java because of its widespread use and because of the good software engineering characteristics that it embodies. We found that all of our language extensions except the implicit **This** type parameter are orthogonal to each other and to Java, which simplifies both their use and implementation. We found that **This** interacts with JL's

constrained parametric polymorphism, which complicates JL's implementation but has little effect on JL's usage. In addition, we found that the class hierarchy optimization interacts with Java's package system, but our design accounts for this interaction.

Our third contribution is to gauge the effectiveness of Java Layers in two evaluations. Our first evaluation compares mixin programming to programming using object-oriented frameworks. Our comparison shows that mixin programming using JL (1) avoids problems of framework evolution and overfeaturing, (2) scales better than frameworks as the number of application features increases, (3) supports a higher level of reuse than frameworks, and (4) supports application variation with more flexibility than frameworks. These results are significant because frameworks are commonly used to build large applications and software product lines. Our second evaluation shows how JL's mixin layers increase code modularity and how this increased modularity can be used to build a software product line from a common code-base. We also define the Sibling design pattern, which coordinates the use of inheritance, nested types, and the most derived types in mixin-generated hierarchies to achieve greater modularity.

Our results are limited in a number of ways. First, we note that mixin layers can only encapsulate features that crosscut their nested classes. If an application designer fails to nest in a mixin layer all the classes a feature affects, then the feature cannot be encapsulated in a layer. Second, the evaluations that we performed were not extensive, real-world trials. Our evaluations contribute to a growing body of evidence that mixins and mixin layers are effective reuse tech-

nologies, but only large-scale experiments can determine their ultimate value. Moreover, JL's semantic checking and class hierarchy optimization have not been implemented, so their utility has not been tested in code. On the other hand, our evaluations do provide evidence that the language features JL has implemented are useful and that they achieve their design objectives. Lastly, JL's research compiler does not implement the complete JL language. In particular, we do not embed JL attributes into class files, so features like constructor propagation and deep conformance have not been tested in non-parametric types.

8.2 Future Work

The success of mixin programming ultimately depends on whether it scales to support large applications over long periods of time, so it is important for future mixin research to include successively larger experiments. These large-scale experiments should demonstrate (1) that mixin programming is an effective and economical way to build real-world software and (2) that mixin tools are mature enough for mission-critical projects. These large-scale experiments should also demonstrate that mixin code remains comprehensible and flexible after intense, long-term maintenance activity. In addition, these experiments should show that teams of programmers can efficiently work on the same mixin application.

As mixin applications become more demanding, mixin tools need to become more robust. A possible successor to JL's source-to-source compiler is a JL-to-bytecode compiler, which would be more production-oriented because of

the output it generates. This new compiler could implement generics using the customized class loader approach described by Agesen, Freund and Mitchell [1]. Under this approach, parametric types are instantiated at load-time, though it would be useful to support both load-time and static instantiation in the same development environment. The JL-to-bytecode compiler would serve as a platform to investigate different bytecode representations for mixins and different approaches to processing them. For example, it would be interesting to investigate whether or not class hierarchy optimization should be performed at load-time.

The most significant impact of a JL-to-bytecode compiler, however, would probably be its effect on mixin language support. In addition to providing an opportunity to more completely implement JL, the new compiler could incorporate into JL feedback from large-scale experiments. For instance, the new compiler could implement JL's semantic checking and then gauge its effectiveness in applications that have hundreds or even thousands of mixins. In addition, the compiler could also be used to explore issues like partial instantiation, user-specified aliases for instantiations, traceability, and debugging support.

A more general topic for future research explores generative programming. Czarnecki and Eisenecker [36] describe C++ as a *two-level language* in which static code is evaluated at compile-time, dynamic code is evaluated at run-time, and both types of code are written in Turing-complete languages (the C++ template language and C++, respectively). C++ templates are arguably too powerful[21], but the idea of splitting compilation into well-defined generative and

[21] Should programmers be able to calculate Fibonacci numbers using template instantiation?

computational phases is intriguing. Of course, the use of macros for many years shows that there is nothing new about pre-processing code. What is new, however, is the idea that the generative phase language deserves as much design attention as the computational phase language. The goal of future research in this area could be the definition of an elegant, meta-programming language for the generative phase of compilation.

Appendix A – Layered Code Micro-Benchmarks

We performed two micro-benchmark tests to determine the effect of deep class hierarchies on Java performance. These tests are not comprehensive, but they do support the arguments made in §2.4.4 for optimizing the class hierarchies of layered applications. The first test addresses method call overhead, and the second test addresses class loading overhead.

All tests were performed on a dual 350 Mhz Pentium II system with 128M of memory. The system runs Windows NT and uses the NTFS file system; hard drive sector size is 512 bytes on an 8G partition. We used Sun's SDK 1.2.2 for our Java environment. The raw data generated by the tests can be found in the optimization specification on the Java Layers web site [57]. All tests were run on optimized class files and garbage collection was turned off.

METHOD CALL TESTS

The method call tests compare method performance in a ten-deep class hierarchy with method performance in a flattened version of the same hierarchy. We construct our deep hierarchy from ten classes that simulate daisy-chained method calls in a moderate-sized, mixin-generated hierarchy. Each class in the hierarchy has a *calc()* method and a *noop()* method. The *calc()* method adds a number to the result of its superclass's *calc()* method, if one exists. The *noop()* method simply calls its superclass *noop()* method, if one exists.

We also construct a flattened class hierarchy that contains only one class. This class defines a *calc()* method and a *noop()* method, which are handcoded ver-

sions of the corresponding leaf class methods in the deep class hierarchy described above. These handcoded methods simulate the inlining performed by the class hierarchy optimization (Chapter 5): *calc()* adds ten numbers and *noop()* does nothing.

The figures below show the execution times of ten million calls to *calc()* and *noop()* in the leaf classes of both hierarchies. Figure 80 shows the results with the just-in-time compiler (JIT) turned off, and Figure 81 shows the results using the JIT. For each test, two primer runs were performed before five actual runs. The results from the five runs were averaged, though the deviation from the mean was never significant.

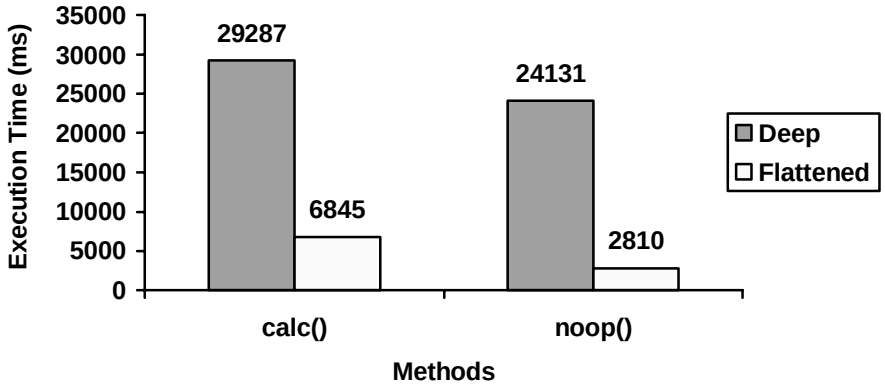


Figure 80 - Method Call Run Times without JIT

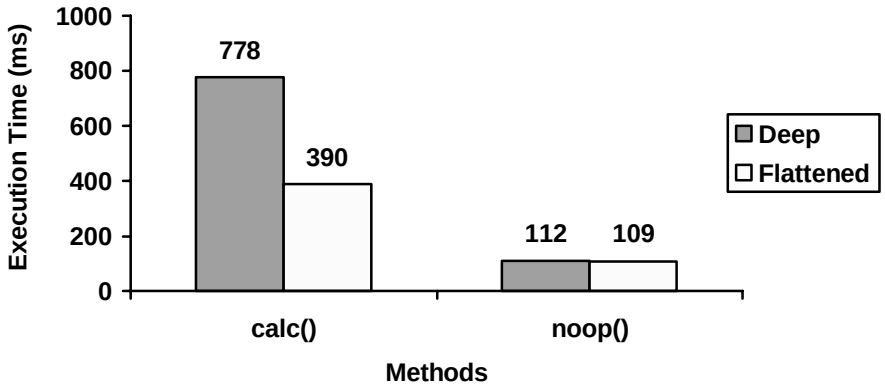


Figure 81 - Method Call Run Times with JIT

With the JIT turned off, flattening speeds up the *calc()* test by a factor of 4.2 and it speeds up the *noop()* test by a factor of 8.6. With the JIT enabled, flattening speeds up the *calc()* test by a factor of 2, but it does not appreciably affect the *noop()* test. The lack of improvement in this last *noop()* test is probably due to JIT optimizations that explicitly eliminate calls to empty methods. Whatever the cause, the results of this last case do not detract from the significance of the other results, which we now discuss.

The non-JIT *noop()* method speedup of 8.6 times can be viewed as the largest possible gain because all non-leaf class processing is eliminated, which removes all overhead. More realistically, the *calc()* method speedup (4.2 for non-JIT, 2 for JIT) is probably the best case improvement that can be expected in practice, since *calc()* is representative of methods that do a little processing in each layer. In general, the more processing performed by each layer, the less important method call overhead becomes and the less dramatic the effect of inlining.

CLASS LOADING TESTS

The goal of the class loading micro-benchmark is to determine how file size affects the load times of Java class files. The micro-benchmark loads two sizes of classes and compares their load times. The small classes contain 500-506 bytes and the large classes contain 2049-2055 bytes. On the test machine, small classes fit into one disk sector and large classes require five sectors. All classes are direct subclasses of `Object` and implement no interfaces. The experiment consists of loading five batches of large and small classes into a Java program.

Figure 82 shows the throughput in kilobytes per second for each batch of large and small classes. The figure indicates that code loads significantly faster when it is packaged in larger files. In each of the five tests, the same number of distinct large and small class files were read from disk and loaded into a Java program. Each test consists of two primer runs and three actual runs. The results of the actual runs were averaged, though the deviation from the mean was never significant.

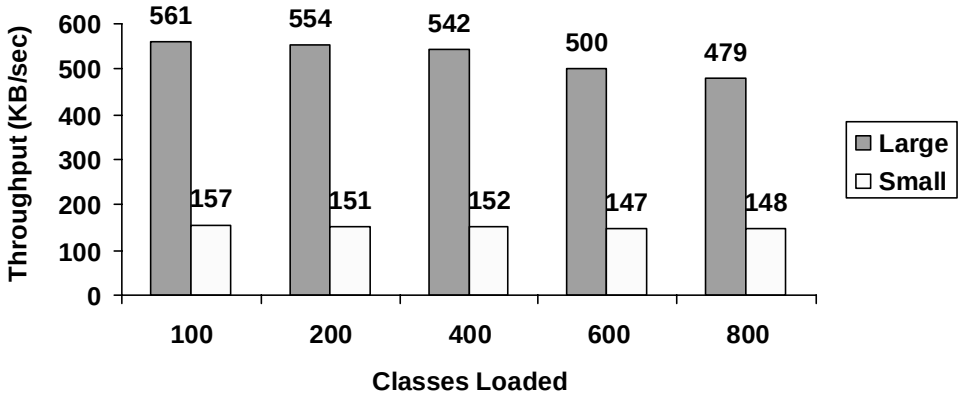


Figure 82 - Load Rate of Different Size Classes

Table 6 shows detailed results for the 100 class and 800 class tests, which represent the extremes in our results. The third and fifth columns normalize data with respect to the small class size results. We see that the number of bytes loaded using large classes is four times greater than the number loaded using small classes, but the total load time for large classes is only 13% to 26% greater. This result indicates that merging classes into larger class files can reduce load time.

Classes Loaded	Bytes Loaded	Normalized Bytes Loaded	Execution Time (ms)	Normalized Execution Time
100 Small	50,600	**1.00**	323	**1.00**
100 Large	204,900	4.05	365	1.13
800 Small	404,800	**1.00**	2,728	**1.00**
800 Large	1,639,200	4.05	3,424	1.26

Table 6 - Detailed Results for Two Load Tests

A more comprehensive study of load times would include a greater variety of class sizes, would use a number of different file systems, and would introduce the effects of networks. Our micro-benchmark, however, is sufficient to validate that loading Java class files is like reading files: In general, less overhead is incurred using fewer large files than using many small files [95].

Appendix B – Regular Expressions in Semantic Checking

In §4.5.2, we described JL's constraint language for semantic checking. In this section, we briefly describe how JL adopts commonly used regular expression syntax for pattern matching whenever possible [42]. Table 7 lists the regular expression meta-characters used in JL.

Meta-Character	Name	Meaning
.	Dot	any one attribute
[..]	attribute class	any attribute listed
[^..]	negated attribute class	any attribute not listed
^	Caret	position at beginning
$	Dollar	position at end
|	Alternation	OR
(..)	Parenthesis	scope delimiters
?, +, *	repetition quantifiers	0 or 1, 1 or more, 0 or more
****	Escape	meta-character escape
!	Mismatch	negate match response

Table 7 - Regular Expression Meta-Characters in JL

JL's regular expression processing differs in one important way when compared to most other languages. The alphabet used in JL to pattern match attribute lists consists of attribute names, not individual characters as in string-based pattern matching. This difference in alphabet means, for instance, that JL attributes must be separated by spaces when they are used in the attribute classes ([..] and [^..]) shown in the table above.

In addition, JL supports a mismatch operator when the exclamation point meta-character (!) appears as the first character in a regular expression. The

mismatch operator causes an expression to return true if no match is found; otherwise false is returned. JL's mismatch operator is similar to the negated return value construct (!~) in Perl [130].

References

1. O. Agesen, S. Freund, and J Mitchell. Adding Type Parameterization to the Java Language. *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 49-65, 1997.

2. A. V. Aho, R. Sethi and J. D. Ullman. *Compilers - Principles, Techniques and Tools*. Addison-Wesley, 1988.

3. D. Ancona, G. Lagorio and E. Zucca. Jam – A Smooth Extension of Java with Mixins. *European Conference for Object-Oriented Programming (ECOOP)*, pages 154-178, 2000.

4. D. Ancona and E. Zucca. A Theory of Mixin Modules: Algebraic Laws and Reduction Semantics. *Mathematical Structures in Computer Science*, to appear.

5. D. Ancona and E. Zucca. A Theory of Mixin Modules: Basic and Derived Operators. *Mathematical Structures in Computer Science*, 8(4):401-446, 1998.

6. ANSI and AJPO. *Reference Manual for the Ada Programming Language*. ANSI/MIL-STD-1815A-1983, February 17, 1983.

7. A. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

8. K. Arnold, J. Gosling and D. Holmes. *The Java Programming Language*, 3rd edition. Addison-Wesley, 2000.

9. M. Arnold, S. Fink, V. Sarkar and P. Sweeney. A Comparative Study of Static and Profile-Based Heuristics for Inlining. ACM SIGPLAN Notices, *Proceedings of ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, 35(7):52-64, January 2000.

10. D. Batory, R. Cardone and Y. Smaragdakis. Object-Oriented Frameworks and Product-Lines. *First Software Product-Line Conference*, pages 227-247, August 2000.

11. D. Batory, G. Chen, E. Robertson and T. Wang. Web-Advertised Generators and Design Wizards. *5th International Conference on Software Reuse*, Victoria, Canada, June 1998.

12. D. Batory, L. Coglianese, M. Goodwill and S. Shaver. Creating Reference Architectures: An Example from Avionics. *Symposium on Software Reusability*, Seattle, Washington, April 1995.

13. D. Batory and B. Geraci. Validating Component Compositions and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering*, pages 67-82, February 1997.

14. D. Batory, B. Lofaso and Y. Smaragdakis. JTS: Tools for Implementing Domain-Specific Languages. *5th International Conference on Software Reuse*, Victoria, Canada, June 1998.

15. D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, pages 355-398, October 1992.

16. D. Batory, V. Singhal, M. Sirkin and J. Thomas. Scalable Software Libraries. *Foundations of Software Engineering (FSE)*, December 1993.

17. E. Berger, B. Zorn and K. McKinley. Composing High-Performance Memory Allocators. *Programming Language Design and Implementation (PLDI)*, pages 114-124, 2001.

18. A. Binstock and J. Rex. *Practical Algorithms for Programmers*. Addison-Wesley, 1998.

19. B. Bokowski and M. Dalm. Poor Man's Genericity for Java. *Java Informations Tage*, Nov. 12, 1998, Frankfurt, Germany.

20. K. Bohrer, A. Christ and B. Rubin. Java and the IBM San Francisco Project. *IBM Systems Journal*, 37(3), 1998.

21. V. Bono, A. Patel and V. Shmatikov. A Core Calculus for Classes and Mixins. *European Conference for Object-Oriented Programming (ECOOP)*, pages 43-66, 1999.

22. J. Bosch. Product Line Architectures in Industry: A Case Study. *International Conference on Software Engineering (ICSE)*, pages 544-554, 1999.

23. G. Bracha and W. Cook. Mixin-Based Inheritance. *Object-Oriented Programming, Systems, Languages and Applications / European Conference for Object-Oriented Programming (OOPSLA-ECOOP)*, pages 303-311, 1990.

24. G. Bracha, M Odersky, D. Stoutamire and P. Wadler. Making the future safe for the past: Adding Genericity to the Java Programming Language. *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 183-200, 1998.

25. A. Brown, R. Cardone, S. McDirmid, and C. Lin. The Specification of the Sibling Design Pattern. *Technical Report CS-TR-02-11*, CS Dept., University of Texas at Austin, 2002.

26. K. Bruce. Increasing Java's expressiveness with ThisType and match-bounded polymorphism. *Technical Report*, Williams College, 1997, http://www.cs.williams.edu./~kim/README.html.

27. K. Bruce, M. Odersky and P. Wadler. A statically safe alternative to virtual types. *European Conference for Object-Oriented Programming (ECOOP)*, pages 523-549, 1998.

28. P. Canning, W. Cook, W. Hill, W. Olthoff and J. Mitchell. F-Bounded Polymorphism for Object-Oriented Programming. *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273-280, 1989.

29. L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys*, 17(4):471-522, December 1985.

30. R. Cardone, D. Batory and C. Lin. Java Layers: Extending Java to Support Component-Based Programming. *Technical report CS-TR-00-11*, Computer Sciences Department, University of Texas (June 2000).

31. R. Cardone, A. Brown, S. McDirmid and C. Lin. Using Mixins to Build Flexible Widgets. *1st International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, April 2002.

32. R. Cardone and C. Lin. Comparing Frameworks and Layered Refinement. *International Conference on Software Engineering (ICSE)*, pages 285-294,

May 2001.

33. C. Cartwright and G. Steel. Compatible Genericity with Run-Time Types for the Java Programming Language. *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 1998.

34. W. Codenie, K. De Hondt, P. Steyaert and A. Vercammel. From Custom Applications to Domain-Specific Frameworks. *Communications of the ACM*, 40(10), October 1997.

35. Comptroller General. Contracting for Computer Software Development. *General Accounting Office report FGMSD-80-4*, 1979.

36. K. Czarnecki and U. Eisenecker. *Generative Programming, Methods, Tools and Applications*. Addison-Wesely, 2000.

37. D. Duggan and C. Techaubol. Modular Mixin-Based Inheritance for Application Frameworks. *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 223-240, 2001.

38. U. Eisenecker, F. Blinn and K. Czarnecki. A Solution to the Constructor Problem of Mixin-Based Programming in C++. Conference on Generative and Component-Based Software Engineering, *Workshop on C++ Template Programming*, Erfurt, Germany, October 2000. Also published in Dr. Dobbs Journal, No. 320, January 2001.

39. K. Fisler, S. Krishnamurthi, D. Batory and J. Liu. A Model CheckingFramework for Layered Command and Control Software. *Workshop on Engineering Automation for Software Intensive System Integration, Monterrey*, California, June 18-22, 2001.

40. M. Flatt, S. Krishnamurthi and M. Felleisen. Classes and Mixins. *Principles of Programming Languages (POPL)*, pages 171-183, January 1998.

41. Free Software Foundation, Boston, USA, http://www.gnu.org.

42. J. Friedl. *Mastering Regular Expressions*. O'Reilly and Associates, 1997.

43. K. Futatsugi, J. Goguen, J. Meseguer and K. Okada. Parameterized Programming in OBJ2. *International Conference on Software Engineering (ICSE)*, pages 51-60, 1987.

44. E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

45. D. Geary. Graphic Java, *Mastering the JFC*, 3rd edition, Sun MicroSystems Press, 1999.

46. W. Gibbs. Software's Chronic Crisis. *Scientific American*, September, 1994.

47. J. Goguen. Abstract Errors for Abstract Data Types. *IFIP Working Conference on Formal Description Programming Concepts*, P. Neuhold (Ed.), MIT Press, pages 370-376, 1979.

48. J. Goguen. Parameterized Programming. *IEEE Transactions on Software engineering*, 10(5): 528-543, September 1984.

49. J. Gosling, B. Joy, G. Steele and G. Bracha. *The Java Language Specification*, 2nd edition. Addison-Wesley, 2000.

50. M. Griss. Implementing Product-Line Features by Composing Aspects. *Proceedings of the 1st Software Product Lines Conference*, pages 271-288, Denver, Colorado, August 2000. Kluwer Academic Publishers.

51. W. Harrison and H. Ossher. Subject-Oriented Programming (A Critique of Pure Objects). *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 411-428, 1993.

52. A. Holub. *Compiler Design in C*. Prentice Hall, 1990.

53. Hyperspace home page at http://www.research.ibm.com/hyperspace.

54. J. Jarvi. Tuples and Multiple Return Values in C++. Turku Center for Computer Science, *Technical Report No. 249*, Finland, 1999. http://www.tucs.fi/Research/Series/techreports.

55. Java 2 Micro Edition, http://java.sun.com/j2me.

56. Java Community Process home page at http://jcp.org.

57. Java Layers home page at http://www.cs.utexas.edu/user/richcar/JavaLayers.html.

58. G. Jimenez-Perez and D. Batory. Memory Simulators and Software Generators. *Symposium on Software Reuse*, pages 136-145, 1997.

59. R. E. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, pages 22-35, June/July 1988.

60. The JOVE compiler from Instantiations, Inc., http://www.instantiations.com.

61. I. H. Kazi, H. H. Chen, B. Stanley and D. Julia. Techniques for Obtaining High Performance in Java Programs. *ACM Computing Surveys*, 32(3):213-240, September 2000.

62. S. Keene. *Object-Oriented Programming in Common Lisp: A Programming Guide in CLOS*. Addison-Wesley, 1989.

63. B. Kernighan and D. Ritchie. *The C Programming Language*, 2nd edition. Prentice Hall, 1988.

64. M. Kersten and G. Murphy. Atlas: A Case Study in Building a Web-Based Learning Environment using Aspect-Oriented Programming. *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 340-352, 1999.

65. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold. An Overview of AspectJ. *European Conference for Object-Oriented Programming (ECOOP)*, pages 327-353, 2001.

66. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier and J. Irwin. Aspect-Oriented Programming. *European Conference for Object-Oriented Programming (ECOOP)*, pages 220-242, 1997.

67. B. Kristensen, O. Madsen, B. Møller-Pedersen and K. Nygaard. Abstraction Mechanisms in the BETA Programming Language. *Principles of Programming Languages (POPL)*, pages 285-298, 1983.

68. J. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers, 2000.

69. H. Lewis and C. Papadimitriou. *Elements of the Theory of Computation*, 2nd edition. Prentice-Hall, 1988.

70. H. Liebermann. Using prototypical objects to implement shared behavior in object-oriented systems. *Object-Oriented Programming, Systems, Languages*

and Applications (OOPSLA), pages 214-223, 1986.

71. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*, 2nd edition. Addison-Welsey, 1999.

72. B. Liskov, A. Synder, R. Atkins and C. Schaffer. Abstraction Mechanisms in CLU. *Communications of the ACM*, 20(8): 564-576, August 1977.

73. M. Lutz. *Programming Python*, 2nd edition. O'Reilly and Associates, 2001.

74. O. Madsen and B. Møller-Pedersen. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 397-406, 1989.

75. K. Maruyama. Automated Method-Extraction Refactoring by Using Block-Based Slicing. *Symposium on Software Reusability*, pages 31-40, 2001.

76. S. McDirmid, M. Flatt and W. Hsieh. Jiazzi: New Age Components for Old Fashioned Java. *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 211-222, 2001.

77. A. Mendhekar, G. Kiczales and J. Lamping. RG: A Case-Study for Aspect-Oriented Programming. *Technical report SPL97-009 P9710044*, Xerox Palo Alto Research Center, February 1997.

78. R. Milner. A Proposal for Standard ML. *Proceeding of the Symposium on Lisp and Functional Programming*, pages 184-197, 1984.

79. D. Moon. Object-Oriented Programming with Flavors. *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 1-8, 1986.

80. I. Moore. Automatic Inheritance Hierarchy Restructuring and Method Refactoring. *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 235-250, 1996.

81. R. Morgan. *Building an Optimizing Compiler*. Butterworth-Heinemann, 1998.

82. A. Myers, J. Bank and B. Liskov. Parameterized Types for Java. *Principles of Programming Languages (POPL)*, pages 132-145, 1997.

83. N. Myhrvold. The Next Fifty Years of Software. *ACM97 Conference*, March 1997. Slides at http://research.microsoft.com/acm97.

84. P. Naur and B. Randell. Software Engineering: Report on a Conference sponsored by the NATO Science Committee, Garmisch, Germany, October 1968. *Scientific Affairs Division, NATO*, January 1969.

85. Object Management Group. *OMG Unified Modeling Language Specification*, version 1.3. OMG 2000. Available at http://www.omg.org.

86. M. Odersky and E. Runne. Measuring the Cost of Parameterized Types in Java. *Research report CIS-98-004*, Advanced Computing Research Centre, University of South Australia, January 1998.

87. W. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. dissertation. Department of Computer Science, University of Illinois at Urbana-Champaign, 1992.

88. H. Ossher, M. Kaplan, A. Katz, W. Harrison and V. Kruskel. Specifying Subject-Oriented Composition. *Theory and Practice of Object Systems*, 2(3):179-202, 1996.

89. K. Ostermann. Dynamically Composable Collaborations with Delegation Layers. To appear in *European Conference for Object-Oriented Programming (ECOOP)*, 2002.

90. Palm Inc., http://www.palm.com.

91. D. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053-1058, December 1972.

92. D. Perry. The Inscape Environment. *International Conference on Software Engineering (ICSE)*, pages 2-11, May 1989.

93. R. Pressman. Software Engineering: *A Practitioner's Approach*, 3rd edition. McGraw-Hill, 1991.

94. D. Roberts, J. Brant and R. Johnson. A Refactoring Tool for SmallTalk. *Theory and Practice of Object Systems (TOPAS)*, 3(4):39-42, 1997.

95. M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems (TOCS)*,

10(1):26-52, February 1992.

96. D. Schmidt. ACE home page: http://www.ece.uci.edu/~schmidt.

97. D. Schmidt. Acceptor and Connector—A Family of Object Creational Patterns for Initializing Communication Services. *Pattern Languages of Program Design 3*, edited by R. Martin, D. Riehle, F. Buschmann and J. Vlissides. Addison-Wesley, 1997.

98. D. Schmidt. An Architectural Overview of the ACE Framework. *USENIX login* magazine, November 1998.

99. D. Schmidt and I. Pyarali. Reactor: An Object Behavioral Pattern for Concurrent Event De-multiplexing and Event Handler Dispatching. *Pattern Languages of Programs Conference*, August 1994. A derivative work by the same title published in *Pattern Languages of Program Design*, edited by J. Coplien and D. Schmidt. Addison-Wesley, 1995

100. V. Singhal. *A Programming Language for Writing Domain-Specific Software System Generators*. Ph.D. Dissertation. Department of Computer Sciences, University of Texas at Austin, August 1996.

101. Y. Smaragdakis. *Implementing Large-Scale Object-Oriented Components*. Ph.D. dissertation, Department of Computer Sciences, University of Texas at Austin, December 1999.

102. Y. Smaragdakis. Interfaces for Nested Classes. *The 8th International Workshop on Object-Oriented Languages (FOOL8)*, London, England, January 20, 2001.

103. Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin Layers. *European Conference for Object-Oriented Programming (ECOOP)*, pages 550-570, 1998.

104. Y. Smaragdakis and D. Batory. Mixin-Based Programming in C++. *Second International Symposium on Generative and Component-Based Software Engineering (GCSE 2000)*, Erfurt, Germany, October 9-12, 2000.

105. J. Solorzano and S. Alagic. Parametric Polymorphism of Java: A Reflective Solution. *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 216-225, 1998.

106. The Standish Group International, West Yarmouth, Massachusetts. CHAOS, 1995. http://www.standishgroup.com.

107. A. Stepanov and M. Lee. *The Standard Template Library*. Hewlett-Packard, 1995.

108. R. Stevens. *UNIX Network Programming*. Prentice-Hall (1990)

109. C. Strachey. Fundamental concepts in programming languages. *Lecture notes for the International Summer School in Computer Programming*, Copenhagen, August 1967.

110. B. Stroustrup. *The C++ Programming Language*, 3rd Edition. Addison-Wesley, 1997.

111. Sun Microsystems, Inc. *Connected, Limited Device Configuration*, specification 1.0a, May 19, 2000.

112. Sun Microsystems, Inc. *Inner Classes Specification*, February, 10, 1997 at http://java.sun.com/products/jdk/1.1/docs/index.html.

113. Sun Microsystems, Inc. *Java 2 Platform Micro Edition (J2ME) Technology for Creating Mobile Devices*, white paper, May 19, 2000.

114. Sun Microsystems, Inc. Java technology, http://java.sun.com.

115. Sun Microsystems, Inc. *Java technology for sealed packages*, http://java.sun.com/products/jdk/1.2/docs/guide/extensions/spec.html.

116. U. Syyid. *The Adaptive Communication Environment: "ACE"*. Tutorial at http://www.cs.wustl.edu/~schmidt/ACE.html.

117. A. Taivalsaari. On the Notion of Inheritance. *ACM Computing Surveys*, 28(3):438-479, September. 1996.

118. P. Tarr, H. Ossher, W. Harrison and S. Stanley. N Degrees of Separation: Multi-Dimensional Separation of Concerns. *International Conference on Software Engineering (ICSE)*, pages 107-119, 1999.

119. S. Thompson. *Haskell, The Craft of Functional Programming*. Addison Wesley, 1996.

120. K. Thorup. Genericity in Java with Virtual Types. *European Conference for Object-Oriented Programming (ECOOP)*, pages 444-471, 1997.

121. F. Tip, C. Laffa, P. F. Sweeny and D. Streeter. Practical Experience with an Application Extractor for Java. *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 292-305, 1999.

122. L. Tokuda. *Evolving Object-Oriented Designs with Refactorings*. Ph.D. Dissertation. Department of Computer Sciences, University of Texas at Austin, December 1999.

123. The TowerJ compiler from Tower Technology, Inc., http://www.twr.com.

124. J. Ullman. Elements of ML Programming. Prentice Hall, 1998.

125. M. VanHilst and D. Notkin. Decoupling Change from Design. *Foundations of Software Engineering (FSE)*, pages 58-69, October 1996.

126. M. VanHilst and D. Notkin. Using C++ Templates to Implement Role-Based Designs. *JSSST International Symposium on Object Technologies for Advanced Software*, Kanazawa, Japan, (March 1996).

127. M. VanHilst and D. Notkin. Using Role Components to Implement Collaboration-Based Designs. *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 359-369, 1996.

128. C. Videira Lopes and G. Kiczales. D: A Language Framework for Distributed Programming. *Technical report SPL97-010 P9710047*, Xerox Palo Alto Research Center, February 1997.

129. P. Wadler, M. Odersky and Y. Smaragdakis. "Do Parametric Types Beat Virtual Types?", unpublished manuscript, posted on Java Genericity mailing list (java-genericity@galileo.East.Sun.com), October 1998.

130. L. Wall, T. Christiansen and J. Orwant. *Programming Perl*, 3rd edition. O'Reilly and Associates, 2000.

131. A. Zaks, V. Feldman and N. Aizikowitz. Sealed Calls in Java Packages. *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 83-92, 2000.

Vita

Richard Joseph Cardone was born in New York City on June 3, 1958, the son of Camille and Pasquale Cardone. After completing public school education in the usual amount of time, he began a college career that was to span twenty-five years. He graduated from the State University of New York at Oneonta in 1983 with a Bachelor of Science degree in Mathematics and a minor in French. He began working at IBM in 1984, a relationship that continues to this day. He earned a Master of Science degree in Computer Sciences from Polytechnic University in Hawthorne, New York, in 1990. He matriculated at the University of Texas at Austin in 1997.

Permanent address: 13533 Anarosa Loop, Austin, TX 78727.

This dissertation was typed by the author.