CHAPTER 33

Using Mixin Technology to Improve Modularity

by Richard Cardone and Calvin Lin

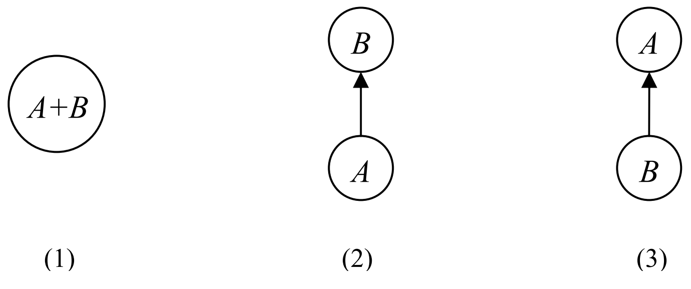
In object-oriented languages, aspects can be defined by generalizing the idea of a class to that of a *mixin*. Mixins, which can be implemented as generic types, become aspects when they contain code for multiple classes. In this chapter, we describe mixins and we explain how mixins can be used to define aspects. We demonstrate the flexibility of mixin programming by prototyping a graphical user interface library that can be configured to run on dissimilar devices. We describe additional language and compiler support that increases the effectiveness of mixin programming. We conclude by proposing some new ideas about how mixins, and generic types in general, can be better supported in object-oriented languages.

33.1. INTRODUCTION

One approach to reducing the cost of software is to make software easier to reuse and, in doing so, to reduce the risk and expense of developing new applications. If existing software can be reused, then the cost of developing and maintaining new code can be significantly reduced. This ability to reuse code depends on two properties: *modularity* and *easy composition*. Modularity allows us to separate concerns [26], making code easier to understand, maintain, and treat as a unit. Easy composition allows us to combine the capabilities of different code modules in different applications.

Unfortunately, today's object-oriented languages, such as Java, are limited in their ability to modularize and compose code. Modularity is limited because the basic unit of encapsulation and reuse is the class. Once the organization of a class hierarchy is fixed, it is always possible to define new features whose implementations *crosscut* the existing set of classes. It is common for features that add global properties, such as security, thread safety, fault tolerance, or performance constraints, to affect code in multiple classes. It is difficult to encapsulate such features in a single class. Instead, object-oriented programs generally consist of sets of collaborating classes [15], and changes to one class often require coordinated changes to others.

Current object-oriented languages are also limited in their ability to compose features. Java's support for composition depends primarily on single inheritance and subtype polymorphism. These mechanisms do not scale well when there are a large number of features. For example, there are three possible ways to organize two features, *A* and *B*, into classes: (1) put them in the same class, (2) make *A* a subclass of *B*, or (3) make *B* a subclass of *A*.



Each choice has different implications regarding the composition of *A* and *B*. For example, the first two choices force *B* to be included whenever *A* is included. Thus, by forcing the programmer to choose a single fixed class hierarchy, Java makes it difficult to compose a collection of features in an orthogonal manner. As the number of features grows, this problem becomes more severe, because the number of possible feature combinations grows rapidly, but the number of feature combinations that can be practically supported does not.

This chapter describes how *mixins* [8], a kind of generic type, can improve the flexibility of Java class hierarchies, thereby improving the ability to modularize code and compose features. The purpose of this chapter is threefold. First, we provide an introduction to mixin programming. We describe our mixin extension to Java and the additional language support that we implemented to increase the effectiveness of programming with mixins. Second, to give a concrete example, we summarize our previously published evaluation [12] of how mixins allow us to build customizable GUI libraries. In particular, we explain how a nested form of mixins, called *mixin layers*

[27][28], can be used to implement a configurable GUI that runs on platforms with widely dissimilar capabilities, such as cell phones, PDAs, and PCs. We also show how mixin layers support the implementation of cross-cutting features, so mixin layers can be thought of as aspects [23]. Third, we propose a new approach of integrating mixins into object-oriented type systems and a new way of reconciling the implementation tradeoffs inherent in parametric polymorphism.

33.2. MIXIN TECHNOLOGY

Our approach to increasing reuse is to build into programming languages better support for modularization and composition. To test this approach, we have developed the *Java Layers* (JL) language [11][20], which extends the compositional capability of Java.

Java Layers extensions include support for constrained *parametric polymorphism* [10] and mixins. Parametric polymorphism allows types to be declared as parameters to code. Parametric polymorphism enhances reuse by allowing the same generic algorithm to be applied to different types; the collection classes in C++'s Standard Template Library [31] are an example of this kind of reuse. JL's implementation is similar to C++'s templates, but in keeping with most proposals [1][9][19] for adding generic types to Java, JL allows type parameters to be constrained.

Mixins are types whose supertypes are specified parametrically. Mixins further enhance reuse over non-mixin parametric polymorphism by allowing the same subtype specialization to be applied to different supertypes. We give an example of mixin reuse in Section 33.2.1.

Mixin layers [27][28] are a special form of mixins that can be used to coordinate changes in multiple collaborating classes. Mixin layers are mixins that contain nested types, which can themselves be mixins. Fidget, our GUI framework described in Section 33.3, is built using mixin layers.

33.2.1 Mixins

The term *mixin* was first used to describe a style of LISP programming that combines classes using multiple inheritance [21][24]. Since then, the mixin concept has evolved to be that of a type whose supertypes are declared parametrically [8][32]. We use the term in this sense and limit ourselves to languages such as Java that support single inheritance of classes. JL supports mixins and other generic types by implementing parametric classes and interfaces.

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. For example, suppose we wanted to extend three unrelated 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.

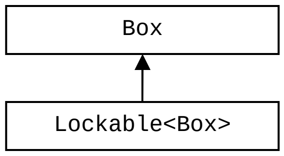
Using mixins, however, we would instead write a single class called `Lockable` that could extend any superclass, and we would instantiate the `Lockable` class with `Car`, `Box`, and `House`. This approach results in only one definition of the lock code. In JL, the `Lockable` mixin would be defined as 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++ 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 precisely 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 its current form, `Lockable`'s capabilities are limited because nothing can be presumed about the type that gets bound to the type parameter `T`. In JL, however, constraints can be specified to restrict the types used in instantiations. For example, the following redefinition of `Lockable` guarantees that `T`'s binding type implements the physical object interface (not shown). This constraint on T means members of `PhysicalObject` can be used within `Lockable` in a type-safe manner.

```
class Lockable<T implements PhysicalObject>
 extends T {…}
```

33.2.2 Stepwise Refinement

The GenVoca software component model [5] provides a conceptual framework for programming with mixins. The model supports a programming methodology of *stepwise refinement* in which types are built incrementally in layers. The key to stepwise refinement is the use of components, called *layers*, that encapsulate the complete implementation of individual application features. Application features are any characteristic or requirement implemented by an application. Stepwise refinement allows custom applications to be built by mixing and matching features.

Mixins implement GenVoca layers. To see how mixins can be used to build applications incrementally, we define the `Colorable` and `Ownable` mixins in the same way that we defined the `Lockable` mixin above. `Colorable` manages a physical object's color, and `Ownable` manages ownership properties. We can now create a variety of physical objects that support various combinations of features:

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

We can think of each of the above instantiations 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, a customized type supporting all the required features is produced. Mixins can be used in this way to provide some of the flexibility of multiple inheritance while avoiding its pitfalls, such as having to manage name collisions and repeated inheritance

[32]. The compositional power of mixins can be further increased when mixin layers are used, which we now discuss.

33.2.3 Mixin Layers

Mixin layers are mixins that contain nested types. A single mixin layer can implement a feature that crosscuts multiple classes. To see how this works, consider an example from our evaluation of customizable GUI libraries, which we call *Fidget* (*F*lexible w*idget*s). Here are simplified versions of the basic 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 {…} …}
```

Figure **33-1** BaseFidget

`BaseFidget` takes no explicit type parameters and we show two of its nested widget classes. In Section 33.3.4, we explain why some parameterized classes don't have explicit type parameters. The main point here, however, is that upon instantiation, the behavior of each of the nested classes in `BaseFidget` is extended by its corresponding class in `ColorFidget`. In this way, feature code scattered across multiple classes is encapsulated in a single mixin layer.

This concludes our introduction to mixin programming; we are now ready to delve more deeply into Fidget's implementation.

33.3. FIDGET DESIGN

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 drove software reuse among general-purpose computers now encourage reuse across different device classes.

To make it easier to reuse code across devices, several standardization efforts are defining new Java runtime environments [18]. These environments are customized for various classes of devices while still remaining as compatible as possible with the Java language, JVM, and existing libraries. For example, Sun’s KVM [29] virtual machine, which is designed to run on devices with as little as 128K of memory, has removed a number of Java language features, such as floating point numbers and class finalization, and a number of JVM features, such as native methods and reflection. In addition, the capabilities of runtime libraries have also been reduced to accommodate limited memory devices. This redesign of the Java libraries leads to two questions that directly concern code reuse and the ability to support crosscutting concerns:

- How does one scale an API to accommodate different devices capabilities?
- How does one reuse the same library code across different devices?

Fidget explores the above issues by implementing a prototype GUI that works on cell phones, Palm OS™ devices [25], 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. Thus, we need a way to encapsulate features that crosscut multiple classes, such as support for color, to a degree that is not possible with standard programming technologies. The goal of Fidget is to test the hypothesis that mixins and mixin layers provide a convenient mechanism for encapsulating crosscutting concerns.

33.3.1 Architecture

Fidget is structured as a stack of the three architectural layers highlighted in Figure 33-2: the Hardware Abstraction layer (HAL), the Kernel layer, and the User layer. On the bottom, the 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.

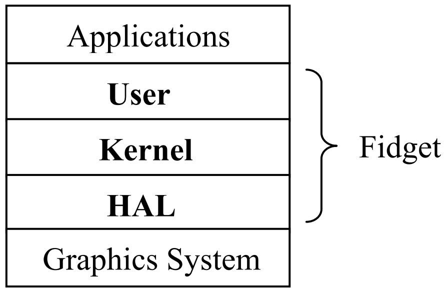


Figure **33-2** Fidget's Architecture

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 [16][30], 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. Even though some optional features cannot be used with all devices, there is only one kernel code-base for all devices.

The Fidget kernel uses a *lightweight* implementation [16] to accommodate devices with constrained memory resources. Lightweight widgets do not have associated peer widgets in the underlying graphics system, which for Fidget is a small subset of either the Java SDK [30] or the J2ME [18] graphic subsystems.[1] 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.

33.3.2 Components

The design of the Fidget kernel classes is based on the `BaseFidget` class introduced in Figure 33-1 in Section 33.2.3. `BaseFidget` provides the minimal implementation for each widget in a nested class. We implemented nine widgets in our prototype; Figure 33-1 shows two of these. These nested widget classes are `Button`, `CheckBox`, `CheckBoxGroup`, `Label`, `Panel`, `TextArea`, `TextComponent`, `TextField`, and `Window`.

[1] For experimental ease, we scaffold Fidget on top of Java instead of writing low-level graphics code for each device.

Optional features are implemented in mixin layers that extend `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 `LightWeightFidget` layer implements lightweight widget support and contains code for most widgets. Fidget's features are listed below.

Table 33-1 Fidget Kernel Mixins

Kernel Mixin	*Multi-Class?*	*Description*
`ButtonSetLabel`	No	Re-settable Button label
`BorderFidget`	No	Draws Container borders
`CheckBoxSetLabel`	No	Re-settable Checkbox label
`TextComponentSetFont`	No	Changeable fonts
`TextFieldSetLabel`	No	Re-settable TextField label
`AltLook`	Yes	Alternative look and feel
`ColorFidget`	Yes	Color display support

Kernel Mixin	*Multi-Class?*	*Description*
`EventBase`	Yes	Basic event listeners/handlers
`EventFidget`	Yes	All event listeners/handlers
`EventFocus`	Yes	Focus event handling
`EventKey`	Yes	Keyboard event handling
`EventMouse`	Yes	Mouse event handling
`LightWeightFidget`	Yes	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. In the next section, we explore the design consequences of defining these superclasses in `BaseFidget`.

33.3.3 The Sibling Pattern

The *Sibling design pattern* uses inheritance relationships between classes that are nested in the same class to enhance code modularity. The pattern itself can be implemented in Java, but mixin layers make it more practical 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 additional JL language support simplifies the solution.

```
class BaseFidget<> {
 public abstract class Component {…}
 public class Button extends Component {…} …}

class ColorFidget<T> extends T {
 public class Component
  extends T.Component {…}
 public class Button
  extends T.Button {…} …}

ColorFidget<LightWeightFidget<BaseFidget>>
```

Figure **33-3** Incorrect BaseFidget

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 33-3, this time showing the widget `Button` and its superclass `Component` (type parameter constraints and most nested classes are not shown). In Fidget, features like color support modify the behavior of `Component` as well as its widget subclasses, as Figure 33-3 shows.

There is, however, a potential pitfall when parent and child classes are nested in the same class. To see the problem, Figure 33-3 also specifies an instantiation of a Fidget GUI with color support. The instantiation includes the `LightWeightFidget` mixin (code not shown), which is structured like `Col-orFidget`.

The class hierarchies generated by the instantiation are shown in Figure 33-4. 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.

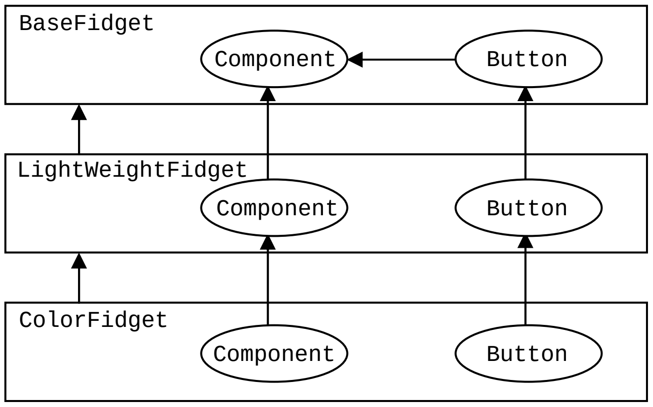


Figure **33-4** Incorrect Hierarchy

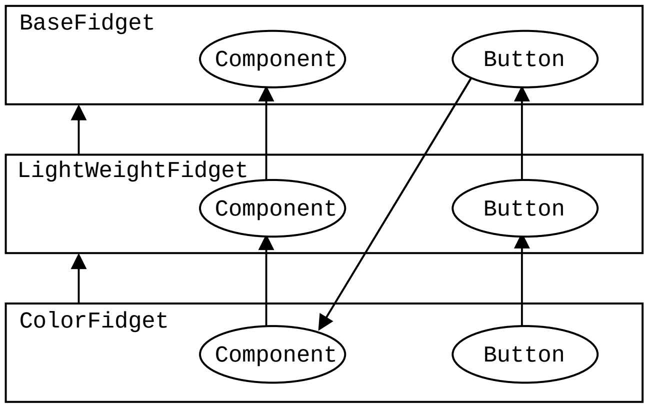


Figure **33-5** Sibling Pattern Hierarchy

The inheritance relationship we really want is shown in Figure 33-5, 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 siblings. In Figure 33-5, `BaseFidget.Button` inherits from the most specialized subclass (`ColorFidget.Component`) of its sibling (`BaseFidget.Component`).

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. By contrast, JL provides a more flexible naming solution that avoids the need for ad-hoc naming conventions; we describe that solution now.

33.3.4 JL's Implicit *This* Type Parameter

JL provides a better way to express the Sibling pattern using its implicit **This** type parameter [11]. Parameterized types in JL have one implicit type parameter and zero or more explicitly declared type parameters. **This** is automatically bound to the leaf class type in a mixin-generated hierarchy, which provides JL with a limited, static, virtual typing [34] capability.

Figure 33-6 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 is `ColorFidget` in our example from Figure 33-3. The redefined `Button` class in Figure 33-6 now inherits from `ColorFidget.Component`.

```
class BaseFidget<> {
 public abstract class Component {…}
 public class Button
  extends This.Component {…} …}
```

Figure **33-6** 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.

33.4. USING FIDGET

To build a Fidget library, we first select the SDK or J2ME Hardware Abstraction layer (HAL) based on the target device's underlying Java support. PC's use the SDK; Palm devices and cell phones use J2ME. As described in Section 33.3.1, the HAL provides a small set of line and curve drawing primitives that is consistent across all platforms.

Next, we specify and compile the features 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 33-2. 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;

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 on which it runs, 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. For example, the definitions for the `Button` and `Window` wrapper classes are:

```
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)
  } }
```

33.5. MIXIN PROGRAMMING SUPPORT

In addition to the implicit **This** type parameter described in Section 33.3.4, JL introduces three other language features and one compiler feature that make mixin programming more effective [11]. In this section, we briefly motivate and characterize each feature.

33.5.1 Constructor Propagation

Superclass initialization is not straightforward in mixin classes because the superclass of a mixin is not known when the mixin is defined [36]. To make constructors convenient to use with mixins, JL introduces the **propagate** modifier for constructors. Constructors are propagated from parent to child class, with constructors marked **propagate** in the parent only able to affect constructors marked **propagate** in the child. (The default constructor in a child class is also considered propagatable.) Constructor propagation is more than the simple inheritance of constructors because constructor signatures and bodies can change when constructors are propagated to child classes.

In Fidget, one measure of the effectiveness of automatic constructor propagation is that many constructors do not need to be hand-coded. In `BaseFidget`, twenty constructors are declared with **propagate**. On average, the thirteen kernel layers that extend `BaseFidget` declare just over one constructor each, which indicates that automatic constructor generation is sufficient in most cases.

33.5.2 Deep Conformance

Mixins provide a powerful way to compose software, but to avoid composing incompatible features, mechanisms are needed to restrict how mixins are used. Type parameter constraints are one mechanism for restricting the use of mixins to avoid incompatibilities. In addition, JL extends the semantics of constrained type parameters to work with the nested structure of mixin layers.

JL's notion of *deep interface conformance* [27] extends Java's idea of interface constraints to include nested interfaces. Normally, a Java class that implements an interface is not required to implement the interface's nested interfaces. JL introduces the **deeply** modifier on `implements` clauses to require classes to implement the nested interfaces of the classes' superinterfaces. In addition, by using the **deeply** modifier on extends clauses, JL also defines *deep subtyping* [27]. Deep subtyping requires that a subtype have the same nested structure as the supertype it extends.

In JL, mixin layers that deeply conform to the same interface, or mixin layers that deeply subtype the same supertype, can be composed with each other because they are compatible at all nesting levels. This compatibility is guaranteed by the compiler.

33.5.3 Semantic Checking

Even with deep conformance, however, undesirable mixin compositions can still be easily created. The ability to restrict how mixins are ordered, or how many times a mixin can appear in an instantiation, requires a higher level of checking than is convenient using OO type systems. We call this extended capability *semantic checking* because mixin compositions should be not only type-safe, but also meaningful.

JL's semantic checking uses semantic *attributes*, which are identifiers or tags chosen by programmers to represent meaningful characteristics in an application. At compile-time, an *ordered attribute list* is associated with each class hierarchy. Attributes are added to lists using **provides** clauses in class definitions. Attribute lists are tested using **requires** clauses in class definitions. These tests use regular expression pattern matching and a count operator to validate the presence, absence, cardinality, and ordering of mixins in a composition. Semantic checking occurs only at compile-time; there is no runtime overhead. JL's semantic checking facility has been specified but has not been implemented.

33.5.4 Class Hierarchy Optimization

JL's programming methodology of stepwise refinement can create deep hierarchies of small classes. The use of many small classes can increase load time and the memory footprint of an application. In addition, stepwise refinement can also increase method call overhead because multiple mixin methods are often called to perform the work of a single method in a conventionally-written application.

At design time, we want the modularity of stepwise refinement; at runtime, we want fast code unimpeded by multiple layers of indirection. By extending existing technology [35] that compresses class hierarchies, we believe the runtime effects of design time layering can be largely eliminated. Our class hierarchy optimization has been specified but has not been implemented.

33.6. Future Work

In this section, we propose two topics for future research. The first topic brings together two prominent and, until now, largely distinct lines of mixin research. The second topic reconciles two approaches to implementing parametric polymorphism in object-oriented languages.

33.6.1 Mixins as Types

Most mixin research falls into one of two categories. The first uses parametric polymorphism to implement mixins by generalizing the idea of a parameterized type. The second defines *mixins as types* and generalizes the idea of a class. In this section, we propose a way to bring these two approaches together.

Using parametric polymorphism, 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++ template classes, which already support mixins. Such research [13][27][36] 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.

On the other hand, mixins can be defined as types that extend their supertypes without relying on parametric polymorphism. The research [3][7][14] 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).

JAM [2] is a recently implemented language that treats mixins as types. 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 { … }
class Child1 = M extends Parent1;
class Child2 = M extends Parent2;
```

Since mixins are types in JAM, `Child1` is a subtype of both `M` and `Par-ent1` 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.

To avoid some of the restrictions of JAM, such as the inability to compose mixins with other mixins, and to support parametric polymorphism, we propose an implementation of parametric types that supports *full* and *partial instantiation*. A parametric type is fully instantiated when all type parameters are bound; a parametric type is partially instantiated when at least one type parameter is unbound. The asterisk (*) is used to indicate unbound type parameters in instantiations.

A key component of this proposal is that all instantiations are types, but only full instantiations can be constructed. The code below depicts mixin `M` and two lines of code that appear outside of `M`. The variable `partial` has type `M<*>`, which is a partial instantiation of mixin `M`. Any full instantiation of `M` can be constructed and assigned to `partial`, as objects of types `M<Parent1>` and `M<Parent2>` below have been.

```
class M<T> extends T { … }
M<*> partial = new M<Parent1>();
partial = new M<Parent2>();
```

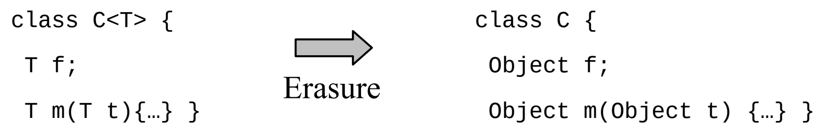
In this proposal, parametric polymorphism (with mixin support) is fundamentally integrated into the type system. In addition, partial instantiation implies partial evaluation: In partial instantiations, members of parametric types that are not dependent on unbound type parameters are accessible.

33.6.2 Implementing Parametric Polymorphism

There are two basic ways to implement parametric polymorphism in object-oriented languages. *Homogeneous* implementations execute the same compiled code for all instantiations of a parametric type. *Heterogeneous* implementations, on the other hand, generate a specialized version of compiled code for each distinct instantiation of a parametric type. In this section, we propose a way to realize the benefits of both approaches in the same implementation.

The homogeneous approach is implemented by Generic Java (GJ) [9] and will be used in future versions of Java [19]. These implementations work by *erasing* type parameters at compile time and replacing them with general types that are appropriate for all instantiations.

Figure 33-7 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.



```
class C<T> {
 T f;
 T m(T t){…} }
```

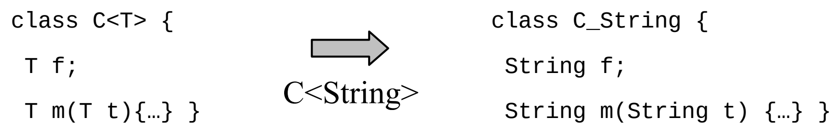
```
class C {
 Object f;
 Object m(Object t) {…} }
```

Figure **33-7** Homogeneous Type Parameter Erasure

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. Type erasure loses information because actual type parameter bindings known at compile time are not available at runtime. This means, for example, that a type parameter cannot specify the (non-array) type in a **new** expression since the actual type is not known at runtime and cannot be allocated. For the same reason, type parameters cannot be used as the types in cast, **catch** or **instanceof** expressions. Most significant, however, is that *homogeneous implementations in Java cannot support mixins* because different mixin instantiations require different supertypes.

Alternatively, the heterogeneous approach is implemented by C++ and Java Layers. JL uses a heterogeneous implementation because mixins can be supported and because of its increased expressiveness, i.e., the ability to use type parameters wherever a type is legal.

Figure 33-8 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. In addition, the substitution of actual type parameters in instantiated types can lead to access control restrictions when the actual type and the parametric type are in different packages [9].



```
class C<T> {
 T f;
 T m(T t){…} }
```

```
class C_String {
 String f;
 String m(String t) {…} }
```

Figure **33-8** Heterogeneous Instantiation

Unfortunately, the choice between homogeneous and heterogeneous implementation affects the programming language and its usage. Homogeneous implementations place numerous restrictions on the use of type parameters; heterogeneous implementations require programmers to consider code size and package placement issues. Rather than lock a language into one approach or the other, we propose to combine the two approaches and to give the programmer control over their use.

Specifically, we propose that parametric types declared with the new **specialize** modifier be instantiated using the heterogeneous approach; otherwise, the homogenous approach is used.[2] Heterogeneously instantiated

[2] Alternately, a **generalize** modifier could be defined, but **specialize** fits in better with the current plan for Java generics.

parametric types can support mixins and the less restrictive use of type parameters. Thus, **specialize** determines how type parameters are used in parametric types. Our proposal gives programmers explicit control over instantiation, whereas programmers currently are implicitly controlled by the implementation choice made by the language.

33.7. RELATED WORK

AspectJ [22][23] is an extension to the Java programming language in which concerns are encapsulated in a new construct called an *aspect*. Aspects implement features that crosscut class boundaries, just as mixin layers do in JL. Both aspects and mixin layers can add new methods to existing classes. Aspects can weave code before or after the execution of a method, an effect JL achieves using method overriding and explicit calls to **super**. Aspects can refine the behavior of any group of existing classes, while mixin layers can only refine the classes nested in their superclasses. Thus, aspects are more expressive and can address more kinds of concerns than JL mixins. On the other hand, aspects must express explicit ordering constraints, while the order of mixin application is implicit in their instantiations. Also, as generic classes, mixins are probably easier to integrate into existing type systems than aspects.

Hyper/J [17] provides Java support for *multi-dimensional separation of concerns* [33]. This approach to software development is more general than that of JL because it addresses the evolution of all software artifacts, including documentation, test cases, and design, as well as code. Hyper/J focuses on the adaptation, integration and on-demand remodularization of Java code. Like JL, encapsulated feature implementations, called *hyperslices* in Hyper/J, can be mixed and matched to create customized applications. Unlike JL, Hyper/J can extract and, possibly, reuse feature code not originally separated into hyperslices. That is, Hyper/J supports the unplanned refactorization of code to untangle feature implementations. While JL generalizes current OO technology, Hyper/J represents a more radical shift in thinking that also requires the development of new composition techniques.

33.8. CONCLUSION

In this chapter, we discussed mixins and mixin layers, and we described how mixin layers implement reusable software components. We summarized an evaluation in which custom GUI libraries are generated using mixin layers. We also described supplemental language support that makes mixin programming easier and more effective.

Additionally, we made two proposals concerning parametric types and mixins. Our first proposal brings together two lines of mixin research by defining partially instantiated parametric types as types. Our second proposal bridges the gap between homogeneous and heterogeneous implementations of parametric polymorphism by giving programmers the choice of implementation when they define parametric types.

33.9. ACKNOWLEDGEMENTS

This work has been supported by NSF CAREER grant ACI-9984660 and a grant from Tivoli.

REFERENCES

[1] Agesen, O., Freund, S., and Mitchell. Adding Type Parameterization to the Java Language. *OOPSLA 1997*.

[2] Ancona, D., Lagorio, G. and Zucca, E. Jam – A Smooth Extension of Java with Mixins. *European Conference for Object-Oriented Programming (ECOOP)*, pages 154-178, 2000.

[3] Ancona, D. and Zucca, E. A Theory of Mixin Modules: Algebraic Laws and Reduction Semantics. *Mathematical Structures in Computer Science*, 12:1-37, 2002.

[4] Arnold, K., Gosling, J., and Holmes, D. The Java Programming Language, 3rd ed. Addison-Wesley, 2000.

[5] Batory, D. and O'Malley, S. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, October 1992.

[6] Batory, D., Singhal, V., Sirkin, M. and Thomas, J. Scalable Software Libraries. *Proceedings of the First ACM Symposium on the Foundations of Software Engineering*, December, 1993.

[7] Bono, V., Patel, A. and Shmatikov, V. A Core Calculus for Classes and Mixins. *European Conference for Object-Oriented Programming (ECOOP)*, pages 43-66, 1999.

[8] Bracha, G., and Cook, W. Mixin-Based Inheritance. *OOPSLA-ECOOP 1990*.

[9] Bracha G., Odersky, M., Stoutamire, D. and Wadler, P. Making the future safe for the past: Adding Genericity to the Java Programming Language. *OOPSLA 1998*.

[10] Cardelli, L. and Wegner, P. On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys* 17, 4, December 1985.

[11] Cardone, R. Language and Compiler Support for Mixin Programming. Ph.D. dissertation, CS Dept., University of Texas at Austin, May, 2002.

[12] Cardone, R., Brown, A., McDirmid, S., Lin, C. Using Mixins to Build Flexible Widgets. *1st International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, April 2002.

[13] Eisenecker, U., Blinn, F., and Czarnecki, K. A Solution to the Constructor Problem of Mixin-Based Programming in C++. *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.

[14] Flatt, M., Krishnamurthi, S. and Felleisen, M. Classes and Mixins. *Principles of Programming Languages (POPL)*, pages 171-183, January 1998.

[15] Gamma, E., Helm, R., Johnson R., and Vlissides, J. *Design Patterns.* Addison-Wesley, 1995.

[16] Geary, D. *Graphic Java, Mastering the JFC*, 3rd ed., Sun MicroSystems Press, 1999.

[17] Hyperspace home page at *http://www.research.ibm.com/hyperspace.*

[18] Java 2 Micro Edition, *http://java.sun.com/j2me*.

[19] Java Community Process, *JSR-14: Add Generic Types to the Java Programming Language*, *http://www.jcp.org*.

[20] Java Layers home page at *http://www.cs.utexas.edu/users/richcar/JavaLayers.html*.

[21] Keene, S. *Object-Oriented Programming in Common Lisp: A Programming Guide in CLOS.* Addison-Wesley, 1989.

[22] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. An Overview of AspectJ. *ECOOP 2001*.

[23] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., and Irwin, J. Aspect-Oriented Programming. *ECOOP 1997*.

[24] Moon, D. Object-Oriented Programming with Flavors. *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 1-8, 1986.

[25] Palm Inc., *http://www.palm.com*.

[26] Parnas, D. On the Criteria to be Used in Decomposing Systems into Modules. *Comm. of the ACM*, 15(12):1053-1058, December 1972.

[27] Smaragdakis, Y. Implementing Large-Scale Object-Oriented Components. Ph.D. dissertation, CS Dept., University of Texas at Austin, December 1999.

[28] Smaragdakis, Y., and Batory, D. Implementing Layered Designs with Mixin Layers. *ECOOP 1998*.

[29] Sun Microsystems, Inc. *Connected, Limited Device Configuration*, specification 1.0a, May 19, 2000.

[30] Sun Microsystems, Inc., Java technology site, *http://java.sun.com*.

[31] Stroustrup, B. *The C++ Programming Language, 3rd Edition*. Addison-Wesley, 1997.

[32] Taivalsaari, A. On the Notion of Inheritance. *ACM Computing Surveys*, Vol. 28, No. 3, Sept. 1998.

[33] Tarr, P., Ossher, H., Harrison, W., and Stanley, S. *N* Degrees of Separation: Multi-Dimensional Separation of Concerns. *ICSE 1999*.

[34] Thorup, K. Genericity in Java with Virtual Types. *ECOOP* (1997).

[35] Tip, F., Laffra C., Sweeney P. and Streeter, D. Practical Experience with an Application Extractor for Java. *OOPSLA* (1999).

[36] VanHilst, M. and Notkin, D. Using C++ Templates to Implement Role-Based Designs. *JSSST International Symposium on Object Technologies for Advanced Software*, Kanazawa, Japan, (March 1996).