Using Mixins to Build Flexible Widgets

Richard Cardone, Adam Brown, Sean McDirmid[†] and Calvin Lin

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712 USA
{richcar, abrown, lin}@cs.utexas.edu

[†]School of Computing
University of Utah
Salt Lake City, UT 84112 USA
mcdirmid@cs.utah.edu

ABSTRACT

When it comes to software that runs on devices as varied as cell phones, PDAs and desktops, one size does not fit all. This paper describes how *mixin layers*, a kind of nested generic type, can be used to implement a graphical user interface library that can be configured to run on platforms with widely dissimilar capabilities. We describe the language support needed to incrementally build software in layers, and we describe how crosscutting concerns can be encapsulated within a layer. We then show how layers can be reconfigured to meet changing requirements. We also show how a new design pattern, the Sibling pattern, can be used with mixin layers to coordinate changes to multiple classes in the same inheritance hierarchy. When used appropriately, the Sibling pattern increases our ability to separate design concerns and to reuse code.

Keywords

Parametric polymorphism, mixin, layers, design pattern, GUI, embedded software.

1. INTRODUCTION

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 [2] runtime environments [23]. 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 [32][34] 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 and the ability to support crosscutting concerns:

> How do we scale an API to accommodate different device capabilities?
> How do we reuse the same library code across different devices?

This paper explores the above questions by designing and implementing a graphical user interface (GUI) that works on cell phones, Palm OS™ devices [29], 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. Our solution uses *mixins* [8] and *mixin layers* [31], along with language support for their use.

The goal of this paper is to test the hypothesis that mixins and mixin layers provide a convenient mechanism for encapsulating crosscutting concerns. We test this hypothesis by building *Fidget*, a flexible widget library, and 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 this paper are as follows:

1. We demonstrate how mixins, supplemented by a number of supporting language features, can be used to build customizable software capable of running on disparate platforms.
2. We define the Sibling design pattern and demonstrate how it can increase code modularity.

AOSD 2002, Enschede, The Netherlands.

3. We add to the growing body of evidence that mixins, mixin layers, and the programming model of layered refinement [5] are effective in increasing code reuse.

This paper proceeds as follows. Section 2 defines the problem in more detail. Section 3 describes our language support for mixins. Section 4 discusses the design of the Fidget library and the components from which it's built. Section 5 describes how Fidget libraries are built and used. Section 6 discusses our approach to component software design. Finally, we present related work and conclusions.

2. THE PROBLEM

This section describes the problem of code reuse in more detail, sketches our solution, and explains how we will evaluate our solution.

In building a graphics library that accommodates dissimilar devices, we would like to mix and match *features* depending on the target execution environment, where a feature is some characteristic or attribute of a GUI 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.

Unfortunately, current programming technologies do not achieve the first goal of completely encapsulating feature implementations. In object-oriented languages like Java, 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 [26][37]. For example, it is common for features that add global properties to a program, such as security, thread safety, fault tolerance, or performance constraints, to affect the code in multiple classes. Generally speaking, object-oriented programs consist of sets of collaborating classes [19], and changes to one class often require coordinated changes to other classes.

In Fidget, for example, color support is an optional crosscutting feature that would break encapsulation and limit reuse if standard object-oriented techniques were used. 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* [21] 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* [21] of feature code in a class makes the class more complex, more interdependent with other classes and, ultimately, more difficult to reuse.

Current object-oriented programming languages such as Java also do not achieve the second goal of making features easy to compose. Java's support for variation depends primarily on single inheritance and subtype polymorphism, which do not scale well when there are a large number of optional features. To understand this shortcoming, consider the possible features that a text field 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—the list goes on. By encapsulating each optional feature in its own class, we could build a text field widget by creating a class hierarchy that contains a base class and selected feature classes in linear order. The result would be a fixed class hierarchy that supports the selected text field features. However, different combinations of features would require different hierarchies. In some cases these new hierarchies would require existing feature classes to have different superclasses, which would lead to a replication of code that quickly becomes unmanageable as the number of different feature combinations increases [6].

2.1 Our Solution

In our solution, Fidget GUIs are constructed by plugging together large-scale components, where each component represents the implementation of a single feature and where each component may contain code for multiple classes. The language we use, Java Layers [13][25], extends the compositional capability of Java to better support large-scale component programming.

Java Layers (JL) extends Java by supporting constrained parametric polymorphism [12] and *mixins* [8]. Parametric polymorphism enhances reuse by allowing the same generic algorithm to be applied to different types. As with most proposals [1][9][24] for adding generic types to Java, JL's implementation of parametric polymorphism differs from C++'s templates [35] by allowing type parameters to be constrained.

Mixins are types whose supertypes are specified parametrically. Mixins further enhance reuse over parametric polymorphism by allowing the same subtype specialization to be applied to different types. In Section 3, we describe JL's support for mixins and its language features that make programming with mixins more convenient.

Mixin layers [30][31] are a special form of mixins that can be used to coordinate changes to multiple collaborating classes. Mixin layers are mixins that contain nested types, which can themselves be mixins. We describe mixin layers in more detail in the next section. In Section 4, we describe Fidget's mixin layers and how they are used to implement the Sibling design pattern, which coordinates changes among collaborating classes and their superclasses.

2.2 Methodology

We evaluate our approach by using 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 also demonstrate that Fidget libraries can be easily configured for cell phones, Palm OS devices, and PCs. We use the 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) [33]. We use the Java 2 Micro Edition (J2ME) Wireless Toolkit 1.0.3 Beta [23] for our cell phone and Palm environments. Our Palm OS tests are run on the Palm OS Emulator version 3.2 [29].

3. SUPPORT FOR MIXINS

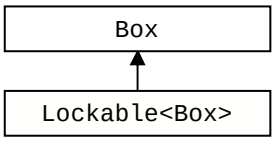
To provide background for the subsequent discussion, this section explains the benefits of programming with mixins, describes the *stepwise refinement* programming methodology, and describes how Java Layers provides language support for stepwise program refinement.

3.1 Mixins

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 wish 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. The lock code would be replicated in three places. With mixins, 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`. The `lock()` and `unlock()` methods would only be defined once. In JL syntax, the `Lockable` mixin would be defined as follows:

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

Mixins are parametric types whose 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`. However, constrained parametric polymorphism restricts the types used in instantiations. For example, the redefinition of `Lockable` below guarantees that `T`'s binding implements the physical object interface (not shown), which means members of that interface can be used within `Lockable` in a type-safe manner. Similar constraints can be specified using an `extends` clause [25].

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

3.2 Stepwise Refinement

The Fidget design is based on the GenVoca software component model [5]. This model encourages a programming methodology of stepwise refinement in which types are built incrementally in layers. The key to stepwise refinement is the use of components, which we call *layers*, that encapsulate the complete implementation of a single feature. Stepwise refinement allows custom applications to be built by mixing and matching the features they need.

Continuing with our previous example, suppose we define the `Colorable` and `Ownable` mixins in the same way that we defined the `Lockable` mixin. `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 [36].

3.3 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 a simplified version 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 {…} …}
```

`BaseFidget` takes no explicit type parameters and contains two nested classes. In Section 4.3, 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.

3.4 Java Layers

Mixins provide a powerful way to compose software, but to avoid composing incompatible features, mechanisms are needed to restrict their use. Type parameter constraints are one mechanism for restricting the use of mixins to avoid incompatibilities. To better support mixin programming, JL extends the semantics of constrained type parameters to work with mixin layers, which we now describe.

JL's notion of *deep conformance* 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. In Figure 1, the use of the **deeply** modifier in JL enforces 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 `Fidget-TkIfc.Button`.

When **deeply** is used in a mixin class's `extends` clause, the superclass's public structure is preserved in the instantiated subclass. By preserved, we mean that if a class nested in a mixin has the same name as a public class nested in the superclass, then the mixin's nested class inherits from the superclass's nested class. In the `ColorFidget` mixin in Figure 1, `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`.

```
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 1 – Deep Conformance

Deep conformance was originally defined by Smaragdakis [30], and the details of its JL implementation are available elsewhere [25]. The actual `ColorFidget` mixin layer contains code for multiple classes and completely implements Fidget's color display support. Deep conformance facilitates Fidget code composition by guaranteeing that a library's nested structure is preserved as new features are added.

Type constraints alone, however, cannot restrict all undesirable mixin compositions. 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 possible using OO type systems. We call this extended capability *semantic checking*. JL's support for semantic checking is part of our ongoing research and is outside the scope of this paper [13][25].

This ends our background discussion of JL. We will describe other JL features as they are encountered during our discussion of Fidget's design.

4. FIDGET DESIGN

This section describes how the Sibling design pattern is implemented using mixin layers and how the notion of *constructor propagation* is useful. To provide context for this discussion, we first discuss Fidget's architecture and its component design.

4.1 Architecture

Fidget is structured as a stack of the three architectural layers highlighted in Figure 2: 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.

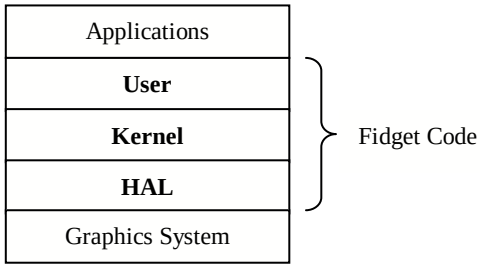


Figure 2 – Fidget's Architecture

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 [20][33], 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.

The Fidget kernel uses a *lightweight* implementation [20] 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.

4.2 Components

We now describe the design of the Fidget kernel classes, which provide the foundation and optional features for all Fidget GUIs. The design is based on the `BaseFidget` class introduced in Section 3.4, 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 `LightWeightFidget` layer implements lightweight

widget support and contains code for most widgets. Fidget's features 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.

4.3 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.

```
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 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 3 above, 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.

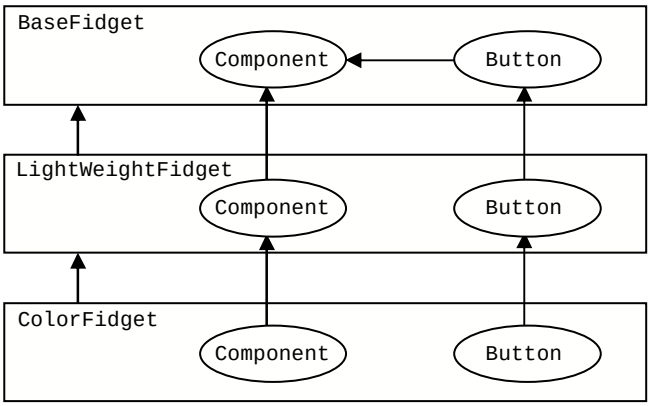


Figure 4 - Incorrect Hierarchy

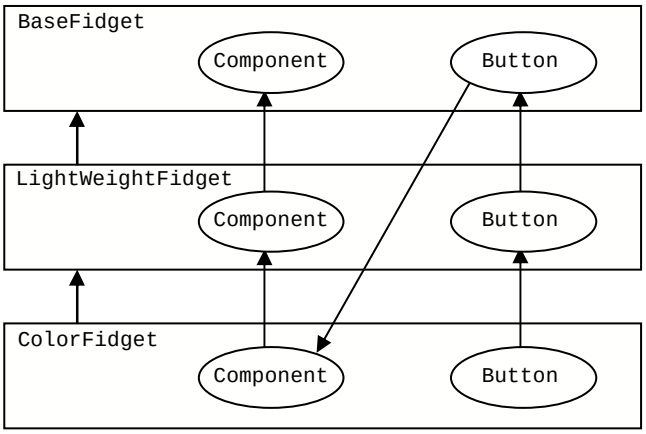


Figure 5 – Sibling Pattern Hierarchy

There is, however, a potential pitfall when parent and child classes are nested in the same class. To see the problem, Figure 3 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 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.

The inheritance relationship we really want is shown in Figure 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 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.

JL provides a better way to express the Sibling pattern using its implicit **This** type parameter [14]. 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 [38] capability.

Figure 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 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 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. This brings us to the last topic in our design discussion, the use of constructors with stepwise refinement.

4.4 Constructor Propagation

Since a superclass does not typically know how to initialize its subclasses, constructors are not inherited in Java and other OO languages. JL, however, encourages the use of small mixin classes that incrementally add function to an application. These mixins often do not require any special initialization, but they do need to initialize their superclasses with the arguments that the superclasses require. In Fidget, for example, the `TextField` class in `BaseFidget` declares four constructors, but no mixin layer that refines `BaseFidget` needs its own initialization for `TextField`. Unfortunately, all layers have to replicate the four `TextField` constructors to make them available at the leaf class. This replication puts a burden on mixin programmers and discourages the use of constructors.

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 in child classes [13].

In Fidget, one measure of the effectiveness of automatic constructor propagation is that many constructors do not need to be hand-coded. In `BaseFidget`, 20 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.

5. USING FIGDET

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 the AWT library. Finally, we give details about the Converter application, which uses Fidget libraries on three different platforms.

5.1 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 2, 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 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 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)
  } }
```

5.2 The Converter Application

As part of the evaluation of Fidget, a simple application named Converter was built on three target devices: JDK 1.3 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 `ConversionPanels`, one with metric units and one with US units. These two `ConversionPanels` 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 version, 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.

6. 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*.

6.1 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 Section 4 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 certainly 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.

6.2 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 do not have all their actual type parameters specified, so they represent a partial application of parameterized types. JL does not support such compositions, though a previous version of the language included a **layerdef** construct [15], which achieved the desired effect through macro expansion.

6.3 Defining the Sibling Pattern

One of the contributions of this paper is the recognition that the inheritance relationship between nested classes described in Section 4.3 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 [4][28], 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 also available [10].

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 Section 4.3, 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 [38].

6.4 Future Work

The goal of Java Layers research is to harness the flexibility of mixins to lower the cost of developing and maintaining quality software. There are two topics related to mixin programming that are the subject of continuing JL research, and we briefly mention them here. The first topic, which we noted in our discussion of deep conformance, concerns the need for higher level semantic checking to limit the way some mixins can be composed. Restricting mixin composition becomes more important as the number of mixins—and the number of incompatible mixin combinations—grows.

The second topic concerns performance, specifically the runtime consequences of generating deeply layered code through mixin instantiation. 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 [39] that compresses class hierarchies, we believe the runtime effects of design time layering can be largely eliminated.

7. RELATED WORK

AspectJ [26][27] 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 [22] provides Java support for *multi-dimensional separation of concerns* [37]. 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 remodularation 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 re-factorization 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.

Jiazzi [28] also encapsulates crosscutting features into components. In Jiazzi, mixin constructions can be expressed that are similar to JL mixins; classes in parameterized components can subclass their component's parameters. Jiazzi components can also implement the *open class pattern*, which incorporates the Sibling pattern described in this paper. The open class pattern is a way for Jiazzi to simulate open classes [16], which allow existing classes to be updated with new methods without

updating the classes' source code. The open class pattern supports separately type checked compilation units, a capability not investigated in JL.

GenVoca [3][4][5][6][30][31] research provides the foundation for JL. JL departs from prior work by concentrating on language support that makes programming with mixins more convenient in current OO languages. Many ideas in JL have precursors in GenVoca implementations. For example, in JTS [4], the Sibling pattern is implemented using well-known names or a less general form of the JL's **This** type parameter.

Dynamic approaches to feature composition exist, including Composition Filters [7] and the use of design patterns such as Decorator and Chain of Responsibility [19]. These approaches are flexible because they compose objects at runtime. JL's static approach to composition, however, allows for off-line constraint checking and optimization, which can reduce the amount of indirection in the code and improve performance.

JL's implicit **This** type parameter combines aspects of both Bruce's *ThisType* [11] and Thorup's virtual types for Java [38]. Both of these approaches require changes to Java's type system and, in the Thorup proposal, increased dynamic type checking. JL's **This**, though less expressive, avoids these complications by limiting its use to parameterized types.

Eisenecker, Blinn and Czarnecki [18] generate customized constructors for C++ mixin classes using generative programming techniques [17]. Their approach uses auxiliary data structures (*configuration repositories*) and programs (*configuration generators*). This auxiliary code requires maintenance when mixins with constructor parameters are composed in new ways or when mixin constructor parameters are changed. JL avoids the cost of maintaining of auxiliary code by providing custom language support.

8. CONCLUSION

This paper 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 GUIs 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.

We have also shown that novel language support in Java Layers, 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.

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. Using the pattern, a base class establishes parent/child relationships between nested sibling classes. Those relationships are then extended to the nested classes in all mixin layers that inherit from the base class. Finally, we have seen how the Sibling pattern can be used to increase code modularity.

9. ACKNOWLEDGMENTS

This work has been supported by NSF CAREER grant ACI-9984660 and a grant from Tivoli. We also thank IBM for their long-term support. We are grateful to Saurabh Boyed for his early Fidget prototype and to the reviewers for their informative and helpful comments.

10. REFERENCES

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

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

[3] Batory, D., Cardone, R. and Smaragdakis, Y. Object-Oriented Frameworks and Product-Lines. *First Software Product-Line Conference*, August 2000.

[4] Batory, D., Lofaso, B. and Smaragdakis, Y. JTS: Tools for Implementing Domain-Specific Languages. *ICSE*, June 1998.

[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] Bergmans, L. The Composition Filters Object Model. *The TRESE Group*, CS Dept., University of Twente, 1994.

[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] Brown, A., Cardone, R., McDirmid, S. and Lin, C. The Specification of the Sibling Design Pattern. *Technical Report CS-TR-02-11*, CS Dept., University of Texas at Austin, 2002.

[11] Bruce, K., Odersky, M. and Wadler, P. A statically safe alternative to virtual types. *ECOOP 1998*.

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

[13] Cardone, R. and Lin, C. Comparing Frameworks and Layered Refinement. *ICSE 2001*.

[14] Cardone, R. and Lin, C. Static Virtual Types in Java Layers. *Technical Report CS-TR-00-25*, CS Dept., University of Texas at Austin, 2000.

[15] Cardone, R., Batory, D. and Lin, C. Java Layers: Extending Java to Support Component-Based Programming. *Technical Report CS-TR-00-11*, CS Dept., University of Texas at Austin, 2000.

[16] Clifton, C., Leavens, G. and Chambers, C. MultiJava: modular open classes and symmetric multiple dispatch for Java. *OOPSLA 2000*.

[17] Czarnecki, K. and Eisenecker, U. *Generative Programming*. Addison-Wesley, 2000.

[18] 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.

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

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

[21] Harrison, W. and Ossher, H. Subject-Oriented Programming (A Critique of Pure Objects). *OOPSLA 1993*.

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

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

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

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

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

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

[28] McDirmid, S., Flatt, M. and Hsieh, W. Jiazzi: New Age Components for Old Fashioned Java. *OOPSLA 2001*.

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

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

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

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

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

[34] Sun Microsystems, Inc. *Java 2 Platform Micro Edition (J2ME) Technology for Creating Mobile Devices*, white paper, May 19, 2000.

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

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

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

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

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