Jiazzi: New-Age Components for Old-Fashioned Java

Sean McDirmid, Matthew Flatt, Wilson C. Hsieh
School of Computing
University of Utah
{mcdirmid,mflatt,wilson}@cs.utah.edu

ABSTRACT

We present Jiazzi, a system that enables the construction of large-scale binary components in Java. Jiazzi components can be thought of as generalizations of Java packages with added support for external linking and separate compilation. Jiazzi components are practical because they are constructed out of standard Java source code. Jiazzi requires neither extensions to the Java language nor special conventions for writing Java source code that will go inside a component. Our components are expressive because Jiazzi supports cyclic component linking and mixins, which are used together in an *open class pattern* that enables the modular addition of new features to existing classes. This paper describes Jiazzi, how it enhances Java with components, its implementation, and how type checking works. An implementation of Jiazzi is available for download.

1. INTRODUCTION

Current Java constructs for code reuse, including classes, are insufficient for organizing programs in terms of reusable software components [26]. Although packages, class loaders, and various design patterns [11] can implement forms of components in ad hoc manners, the lack of an explicit language construct for components places a substantial burden on programmers, and obscures a programmer's intent to the compiler or to other programmers. As object-oriented software systems increase in size and complexity, components are becoming central to the design process, and they deserve close integration with the language.

Components should support separate compilation, which enables development of large programs and deployment of components in binary form, and external linking, which eliminates hard-coded dependencies to make components as flexible as possible for client programmers [7]. In addition, components integrated into a class-based language, such as Java, should also fit well with the class system:

- R1. Components should import classes that can be both instantiated and subclassed within the component. Inheritance across component boundaries is necessary for grouping classes and class extensions into reusable components.
- R2. Components should accept imported classes that supply more methods than the component requires or expects. Requiring an exact match on methods of an imported class would prohibit the composition of class-extending components; e.g., *mixins* [4].

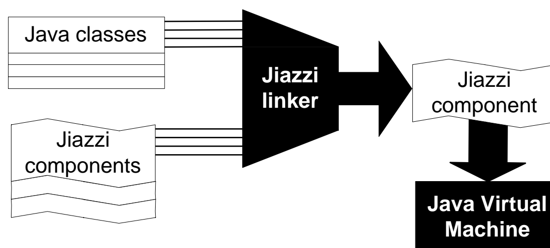

Figure 1: Jiazzi components are constructed from Java classes and other components, and can be loaded in Java Virtual Machines for execution.

- R3. Cyclic component linking should be allowed to resolve mutually recursive dependencies among components. Mutually recursive "has a" relationships are especially common at the class level, and naturally span component boundaries.

Jiazzi is our new component system for Java based on program units [10]. Jiazzi provides the first combination of components and classes that supports all of the above features. Figure 1 illustrates at a high level how Jiazzi works: a Jiazzi component can be built from Java classes and other Jiazzi components. The resulting component can execute directly on a Java Virtual Machine.

Jiazzi makes two contributions to component programming in Java that also apply to other statically typed, object-oriented languages. First, no special core language extensions or conventions need to be used in the Java code used to construct a component. Instead, Jiazzi integrates with Java using a stub generator and an external linker. Because subclassing across component boundaries and cyclic component linking is supported, component boundaries can be placed in the design naturally. This also allows easy retrofitting of legacy Java code into component-based designs.

Second, Jiazzi can support the addition of features to classes without editing their source code or breaking existing class variant relationships. Such functionality is already provided by languages that support open classes [6]. A combination of mixins and cyclic component linking is used to simulate open classes with what we call the *open class pattern*. Using the open class pattern in Jiazzi provides a solution to the extensibility problem [7], which arises from the tension between adding features to and creating variants of a class. With the open class pattern, we can replace the use of many design patterns used to implement modular feature addition,

such as abstract factories and bridges, with a combination of external linking and Java's in-language constructs for subclassing and instantiation.

Section 2 gives an overview of Jiazzi components and shows how they can be used in program designs. Section 3 describes how Jiazzi can be used to modularly add features to classes with mixins and the open class pattern. Section 4 explains how type checking in the presence of separate compilation works in Jiazzi. Section 5 describes our implementation of Jiazzi and the interactions between Java and Jiazzi. Section 6 discusses related work. Section 7 discusses future work, and summarizes our conclusions.

2. OVERVIEW

Components in Jiazzi are constructed as *units* [10]. A unit is conceptually a container of compiled Java code with support for "typed" connections. There are two types of units: *atoms*, which are built from Java classes (including Java interfaces), and *compounds*, which are built from atoms and other compounds.

Units import and export Java classes. Classes imported into a unit are exported from other units; classes exported from a unit can be imported into other units. Linking specified by compounds determines how connections are made between exported and imported classes. Groups of classes are connected together when units are linked; we call these groups of classes *packages* to emphasize their similarity to packages in standard Java. Using package-grained connections reduces the quantity of explicit connections between units, which allows the component system to scale to larger designs.

Jiazzi includes a component language that provides a convenient way for programmers to build and reason about units. Using this language, the structure of classes in a unit's imported and exported packages can be described using *package signatures*. Because package signature can be used in multiple unit descriptions, they enhance the component language's scaling properties.

We introduce Jiazzi using a simple example that composes a user interface (UI) library with an application into a complete program. Because they are used to describe units, we will first describe package signatures.

2.1 Package Signatures

Package signatures are constructs that are used to describe the visible structure of classes in a Java package. In Figure 2, the package signature ui_s describes a UI library with classes Widget, Button, and Window; the package signature applet_s describes an application with class Program. In the package signature the structure of a class is described using a *class signature*. The class signature of Window in ui_s specifies that the class has the superclass ui_p.Widget and has the public methods ***add*** and ***show***. In our example, only the methods and superclasses of classes are described, but class signatures can also describe interface subtyping and class members such as fields, constructors, and inner classes. Class signatures can also describe Java interfaces as well as class and member modifiers (e.g., **protected**, **abstract**).

Class signatures are parameterized by the enclosing package signature's *package parameters*, which must be bound to packages when the package signature is used. The only package parameter of ui_s is ui_p. We assume **Object** and **String** are built-in for the purposes of this example, which also reflects the close coupling of these classes to the Java Virtual Machine (see Section 5 for more details). Classes other than **Object** and **String** must be referred to through one of the package signature's package parameters. In ui_s, the direct superclass of Window is specified as ui_p.Widget, which only comes from the same package as Window if ui_p is bound to the same package that provides Window. Allowing a package to implicitly reference itself would limit the package signature's use; using the open class pattern in Section 3 depends on the flexibility of package signatures that do not implicitly use self-reference.

file: ./ui_s.sig

```
signature ui_s<ui_p> {
 class Widget extends Object
 { void paint(); }
 class Button extends ui_p.Widget
 { void setLabel(String); }
 class Window extends ui_p.Widget
 { void add(ui_p.Widget); void show(); }
}
```

file: ./applet_s.sig

```
signature applet_s<ui_p> {
 class Program extends ui_p.Window
 { void run(); }
}
```

Figure 2: Package signatures ui_s, which describes a UI library, and applet_s, which describes an application; as conventions in this example, package signature names end with _s, and package parameters end with _p.

file: ./applet.unit

```
atom applet {
 import ui_in : ui_s<ui_in>;
 export app_out : applet_s<ui_in>;
}
```

Figure 3: An atom *applet* that imports a UI library and exports an application; as conventions in this example, the names of imported packages end with _in, and the names exported packages end with _out.

2.2 Atoms

The atom *applet* shown in Figure 3 imports Java classes in the package ui_in that implement a user interface library described by package signature ui_s, and exports classes in a package app_out that implement an applet described by package signature applet_s. Within a unit, the package parameters in the package signatures used to describe each imported and exported package must be bound only to the unit's imported and exported packages. Therefore, class signatures of imported and exported classes only refer to the unit's imported and exported classes. For example, after package signature applet_s is used in *applet*, the superclass of app_out.Program is ui_in.Window because the package parameter ui_p is bound to ui_in.

A unit's declarations of imported and exported packages constitutes its *unit signature*. Class signatures provided by the unit signature are necessary to implement separate type checking in Jiazzi. Inside a unit, the implementation of the unit's imported classes are not

file: ./applet/app_out/Program.java

```
package app_out;
public class Program
 extends ui_in.Window {
 ui_in.Button b = new ui_in.Button();
 public  Program() {
  b.setLabel("start"); add(b);
 }
 public void run() { show(); }
}
```

Figure 4: The Java source implementation of app_out.Program in atom *applet*.

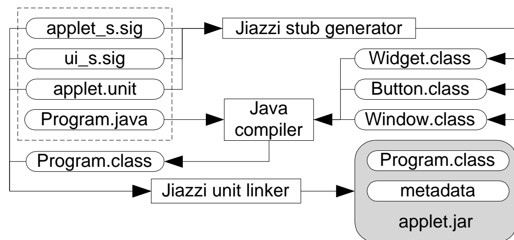

Figure 5: The files and development process of building *applet*; files with source shown in other figures are in the dashed rectangle, tools are in rectangles, files are in rounded rectangles, archive files are shaded rounded rectangles.

visible; outside the unit, the implementation of the unit's exported classes are not visible. We explain separate type checking in more detail in Section 4.

Atoms are built from Java classes that can be compiled from normal Java source code. Shown in Figure 4 is the Java source for *applet*'s exported class app_out.Program. The Java source can instantiate and subclass imported classes. For example in the implementation of app_out.Program, the class ui_in.Window can be subclassed and the class ui_in.Button can be instantiated. Java source can only refer to imported classes, exported classes, or private classes contained in the atom.

Figure 5 shows how the atom *applet* is developed in our implementation of Jiazzi. Files provided by the developer, unit definitions, package signatures, and Java source, are located in separate source files. Since the implementation of imported classes are unavailable, standard Java source compilers (e.g., javac or jikes) cannot automatically know about the structure of imported classes. For this reason, our implementation provides a *stub generator* that uses the class signatures of imported classes to generate stub class files. In our example, stub class files are generated for the imported user interface classes in package ui_in. These class files are then used to compile Program.java into Program.class using a standard Java source compiler.

After Java source compilation, the Jiazzi component linker performs type checking to ensure that the atom's compiled Java code conforms to its unit signature. The class files for classes contained in the atom are placed into a Java archive (JAR) file, which is the atom's *binary form*. For example, Program.class is placed into the atom *applet*'s binary form, applet.jar. The atom's unit signature is also placed into the JAR file as component meta

file: ./ui.unit

```
atom ui {
 export ui_out : ui_s<ui_out>;
}
```

file: ./linkui.unit

```
compound linkui {
 export ui_out : ui_s<ui_out>,
        app_out : applet_s<ui_out>;
} {
 local u : ui, a : applet;
 link u@ui_out to a@ui_in, u@ui_out to ui_out,
      a@app_out to app_out;
}
```

Figure 6: An atom *ui* that exports a UI library, a compound *linkui* that links atoms *ui* and *applet* together.

data. More information about developing with Jiazzi can be found in the user manual [1].

2.3 Compounds

The atom *ui* in Figure 6 exports a package of classes that implement the user interface library described by package signature ui_s from Figure 2. The compound *linkui* in Figure 6 links this atom to the atom *applet* from Figure 3. The unit signature of a compound has the same form as that of an atom; *linkui* exports packages described by the package signatures from Figure 2. Following its unit signature is the compound's link section. In the link section, the Java classes contained in units are conceptually copied by instantiating the units into *unit instances* using the **local** statement. In *linkui*, the atoms *applet* and *ui* are respectively instantiated into the unit instances *a* and *u*.

The **link** statement makes connections from source packages on the left to sink packages on the right of each **to** clause. A source package is either an imported package of the compound or an exported package of one of the compound's unit instances. A sink package is either an imported package of one of the compound's unit instances or an exported package of the compound. We write *v*@p as the notation for the imported or exported package p of unit instance *v*.

In the compound *linkui*, the exported package *u*@ui_out is connected to the imported package *a*@ui_in. The meaning of this connection is that all references to classes in the package ui_in are replaced with references to classes in *u*@ui_out in the unit instance *a* using name equivalence of the unqualified class name. For example, references to ui_in.Widget inside the implementation of classes in unit instance *a* are replaced with references to Widget.ui_out in *u*.

An exported package of a unit instance is available outside of the linking compound if it is connected to one of the compound's exported packages. Encapsulation at the component level is hierarchical; linking compounds are only aware of the compound's unit signature, and are unaware of the units that initially provided the exported packages. In *linkui*, the exported package *u*@ui_out is connected to the package ui_out that is exported by the compound. Compounds that instantiate *linkui* can use this exported package, but will not know that these classes are initially exported

Figure 7: A graphical illustration of the connections made by `linkui`.

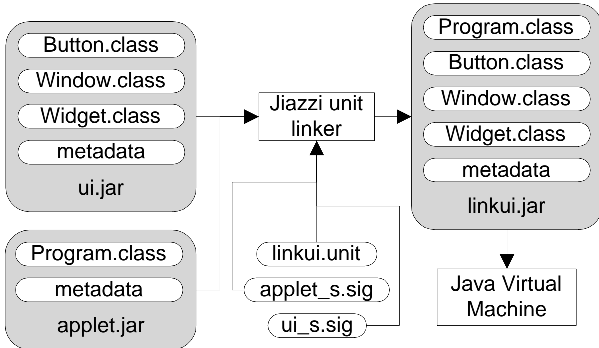

Figure 8: The files and development process of building and executing `linkui`.

by the atom `ui`.

An illustration of `linkui`'s linking is shown in Figure 7. Unit instances and the enclosing compound are represented as alternately shaded rounded rectangles. Packages are represented as boxes that are black tabbed on the left and rounded on the right. Imported packages come into a unit from the left, while exported packages leave the unit from the right. Connections are represented as lines from the right rounded part of a package to the left black tabs of other packages.

The same unit can be used to create multiple unit instances, each of which can be used in different contexts. The imports of each unit instance can be connected differently, and each unit instance exports a distinct group of classes. There is no restriction on the number of unit instances, within a single compound or complete program, that can be created using the same unit.

Informally, a compound can be reduced to an atom by:

1. Copying the (reduced) bodies of all units used to create a unit instances;
2. Concatenating all of the copied bodies, and renaming class references according to the mapping specified by the link section; this rewriting is analogous to the way that a linker finds and updates offsets at link time.

The result can itself be used to create unit instances that undergo further linking within larger compounds. Of course, there is no guarantee that the concatenated bodies are well-formed unless some form of checking has been applied to units during linking. We explain these rules in Section 4.

The development of `linkui` is shown in Figure 8. Both atoms `applet` and `ui` must be linked into their binary forms before

file: ./color_s.sig
```
signature color_s<orig_p> {
 class Widget extends orig_p.Widget
 { void setColor(int); }
 class Button extends orig_p.Button {}
 class Window extends orig_p.Window {}
}
```
file: ./font_s.sig
```
signature font_s<orig_p> {
 class Widget extends orig_p.Widget
 { void setFont(int); }
 class Button extends orig_p.Button {}
 class Window extends orig_p.Window {}
}
```
file: ./both_s.sig
```
signature both_s<orig_p> {
 class Widget extends orig_p.Widget
 { void setColor(int); void setFont(int); }
 class Button extends orig_p.Button {}
 class Window extends orig_p.Window {}
}
```
file: ./nop_s.sig
```
signature nop_s<orig_p> {
 class Widget extends orig_p.Widget {}
 class Button extends orig_p.Button {}
 class Window extends orig_p.Window {}
}
```

Figure 9: Package signatures `color_s`, `font_s`, and `both_s` that describe packages which respectively add color, font, and both color and font features to the package parameter `orig_p`, and `nop_s`, which is an empty extension of `orig_p`.

`linkui` can be linked. The Jiazzi component linker performs type checking and copies the class files from each atom, which are rewritten according to how connections are made in the compound. In our example, the linker creates the JAR file `linkui.jar`, which is `linkui`'s binary form. The class file `Program.class` is copied from `applet.jar` into `linkui.jar`. Since `u@ui_out` is connected to `a@ui_in`, `Program.class`'s references of imported classes in the package `ui_in` are changed in the copy to be references of the exported classes in `u@ui_out`, which are also copied into `linkui.jar`. The format of a compound's binary form is the same as an atom's binary form; after linking there is no distinction between atoms and compounds. Since the compound `linkui` has no imports, its classes can safely be executed in a Java Virtual Machine by placing `linkui.jar` in the classpath.

3. FEATURE EXTENSIBILITY

In addition to decomposing a design into many classes, it is also useful to decompose a design into multiple features [21]. Features cross cut class boundaries and benefit from being implemented in separate components [13]. To demonstrate Jiazzi's expressiveness, we show how Jiazzi can be used to decompose class library features into multiple components. We continue with our example of a UI library by adding the color and font feature to the UI library using the package signatures in Figure 9. We present two approaches: the pure mixin approach, which utilizes mixins to add features to classes, and the open class pattern, which is an improvement of the pure mixin approach that uses cyclic linking to solve the extensibility problem.

file: ./mix.color.unit
```
atom mix.color {
 import ui_init : ui_s<ui_init>,
        ui_in : nop_s<ui_init>;
 export ui_out : color_s<ui_in>;
}
```
file: ./mix.font.unit
```
atom mix.font {
 import ui_init : ui_s<ui_init>,
        ui_in : nop_s<ui_init>;
 export ui_out : font_s<ui_in>;
}
```
file: ./mix.both.unit
```
compound mix.both {
 import ui_init : ui_s<ui_init>,
        ui_in : nop_s<ui_init>;
 export ui_out : both_s<ui_in>;
} {
 local c : mix.color, f : mix.font;
 link ui_init to c@ui_init,
      ui_init to f@ui_init, ui_in to c@ui_in,
      c@ui_out to f@ui_in, f@ui_out to ui_out;
}
```

Figure 10: Units `mix.color`, `mix.font`, and `mix.both`, which use mixin constructions to add color, fonts, and both color and fonts to a package of UI library classes.

3.1 Mixins

Units are powerful enough to express a kind of mixin [4], where an exported class subclasses an imported class. Such an exported class will have all methods present in the actual class connected to the imported superclass: if a method ***m*** is visible in a class imported into a unit, then outside of the unit ***m*** is visible in any exported class that subclasses the imported class, even if ***m*** is not visible in the imported class's signature within the unit.

To use mixins in feature addition, suppose we are writing a unit that adds a feature to a single package of classes. The unit must import an "initial" construction of the classes before any features are added, which we call the *init-package*. The init-package establishes variant relationships and provides initial functionality. The unit also imports the "previous" construction of the classes that are the result of the last feature added, which we call the *in-package*. The in-package is an extended version of the init-package. The unit exports an *out-package*, which is the extended version of the in-package: each class in the out-package subclasses a class in the in-package with the same unqualified name, forming a series of mixins. The features added by the unit are added to classes in the out-package.

Figure 10 uses mixins in the atom `mix.color` to add the color feature to a package of classes that implement a UI library. The imported packages `ui_in` and `ui_init` and the exported package `ui_out` are a UI library's in-, init-, and out-packages, respectively. The package signature `ui_s` from Figure 2 describe the UI library's initial structure. The package signature `nop_s` is used to describe the imported package `ui_in` as an extension of `ui_init` without any new methods. The package signature `color_s` from Figure 9 adds the new method ***`setColor`*** to class `Widget` and establishes normal mixin relationships between classes in `ui_out` and `ui_in`.

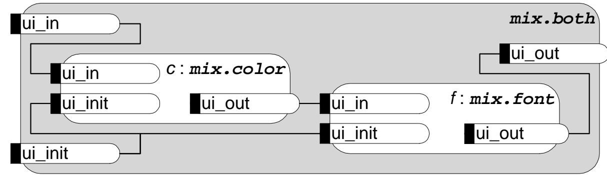

Figure 11: A graphical illustration of connections made in the compound `mix.both`.

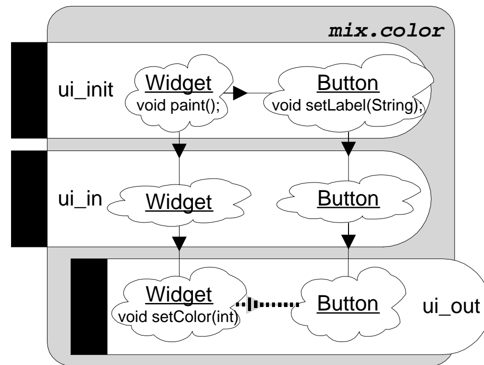

Figure 12: A graphical illustration of the subclassing relationships local to the atom `mix.color` in Figure 10; black arrows point to subclasses, the grey-dashed arrow points to a desired subclassing relationship that cannot be achieved using mixins alone.

The atom `mix.font` in Figure 10 adds the font feature to a UI library in the same way that the atom `mix.color` adds the color feature. The compound `mix.both` composes both `mix.color` and `mix.font` together to create a unit that adds both the font and color features (using package signature `both_s` from Figure 9) to a UI library. Inside `mix.both`, the method ***`setColor`*** is visible in the class `ui_out.Widget` exported from the unit instance *f* because it is a subclass of `ui_out.Widget` exported from the unit instance *c*. This allows both methods ***`setColor`*** and ***`setFont`*** to be visible in the class `ui_out.Widget` exported from `mix.both`. The linking is illustrated in Figure 11.

Mixins in Jiazzi enable reuse of class implementations only; they do not provide a common type to describe the functionality they add, unlike language-level mixin proposals such as the Java language extension JAM [2]. Jiazzi mixins address a different design space: they are link-time abstractions that enable transparent class inheritance across component boundaries, as opposed to abstractions in the core language that enable fine-grained mixin-oriented programming.

problematic as we can see in Figure 12, which shows the subclassing relationship of `Widget` and `Button` inside *`open.color`* (the subclassing relationships of `Window` are similar to `Button`'s). Subclassing is used to make the class `Button` a variant of `Widget` by having `ui_init.Button` subclass `ui_init.Widget`. The feature of color is added to `Widget` in the class `ui_out.Widget`. Using mixins fails, however, because we cannot combine classes `ui_out.Button` and `ui_out.Widget` to create "color buttons." The problem that occurs when trying to add features (a.k.a. *vertical class extension*) and create variants (a.k.a. *hori-*

file: ./open.color.unit
```
atom open.color {
 import ui_in : ui_s<ui_fixed>,
        ui_fixed : nop_s<ui_out>;
 export ui_out : color_s<ui_in>;
}
```

Figure 13: An atom `open.color` that uses the open class pattern to add color feature to a UI library.

zontal class extension) is known as the *extensibility problem* [7]. We can solve this problem by using an approach that utilizes cyclic component linking as well as mixins.

3.2 Open Class Pattern

A general solution to the extensibility problem must not only allow the modular addition of new features to existing classes; it must also ensure that added features are visible in all variants of the updated class. For example, the new method ***setColor*** added to `Widget` must be visible in instances of `Widget`'s variant `Button`. *Open classes* in MultiJava [6] satisfies this requirement when features are new methods. Jiazzi does not directly support open classes, but the *open class pattern* utilizes Jiazzi's expressive linking facilities to simulate open classes.

The open class pattern utilizes mixins and "upside-down" mixins, which behave the same as normal mixins, just from a different perspective: imported classes subclass exported ones. Method visibility is the reverse as that for mixins: if a method ***m*** is visible in a class exported from a unit, then inside of the unit ***m*** is visible in any imported class that subclasses the exported class, even if ***m*** is not visible in the exported class's signature outside the unit. Using "upside-down" mixins necessarily requires cyclic component linking.

To apply the open class pattern, suppose we are writing a unit that adds a feature to a single package of classes. Like the pure mixin approach, the unit must import an in-package and export an out-package. The new feature is still implemented in the out package's classes, which is the extended version of the in-package (forming normal mixins). The key to the open class pattern is that instead of importing an initial-package, the unit instead imports a *fixed-package*, which is the result of all features applied to the package of classes. The fixed-package is the extended version of the out-package (forming the "upside-down" mixins).

Figure 13 uses the open class pattern in the atom *open.color* to add the color feature to classes in a UI library. The imported packages `ui_in` and `ui_fixed` and the exported package `ui_out` are the UI library's in-, fixed-, and out-packages, respectively. The package signature `ui_s` from Figure 2 is used to describe the structure of `ui_in`. The package signature `nop_s` from Figure 9 is used to establish the "upside-down" mixin relationship between `ui_fixed` and `ui_out`. As with the pure mixin approach, the new method ***setColor*** is added to the class `Widget` in `ui_out`.

Figure 14 shows the subclassing relationship of `Widget` and `Button` inside *open.color* (again, the subclassing relationships of `Window`'s is like that of `Button`). Only classes in the fixed-package should be instantiated or subclassed to create variants. For example, `ui_fixed.Widget`, not only `ui_in.Widget`, is a superclass of `ui_in.Button` since `Button` is a variant of `Widget`. As can be seen in the figure, the class `ui_out.Widget`, which adds the new method ***setColor***, is inserted as a superclass of `ui_fixed.Widget`, which establishes the "upside-down" mixin relationship.

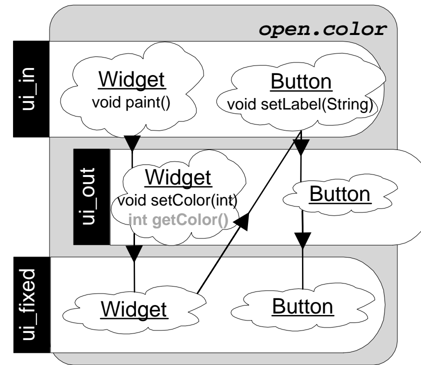

Figure 14: A graphical illustration of the subclassing relationships local to the atom `open.color` in Figure 13.

file: ./open.color/ui_out/Widget.java
```
package ui_out;
public class Widget
 extends ui_in.Widget {
 private int clr;
 public void setColor(int c) {
  clr = c;
 }
 protected int getColor() {
  return clr;
 }
}
```
file: ./open.color/ui_out/Button.java
```
package ui_out;
public class Button
 extends ui_in.Button {
 public void paint() {
  ... = this.getColor() ...;
  super.paint();
 }
}
```

Figure 15: The Java source implementations of `ui_out.Widget` and `ui_out.Button` in `color`.

Shown in Figure 15 is the Java source for classes `Widget` and `Button` in package `ui_out`. The method ***getColor*** is hidden outside of the atom *open.color* by `ui_out.Widget`'s class signature. Because of `ui_fixed.Widget`'s "upside-down" mixin relationship with `ui_out.Widget`, inside *open.color*'s implementation the method ***getColor*** is visible in `ui_out.Button`, as shown in the Java source for `ui_out.Button`.

Figure 16 shows how multiple feature-adding components can be combined into one feature-adding component using a compound. The atom *open.font* uses the open class pattern to add the font feature to a UI library. The compound *open.both* instantiates both *open.color* and *open.font* and applies the resulting unit instances to its in-package import of the UI library. The com-

file: ./open.font.unit

```
atom open.font {
 import ui_in : ui_s<ui_fixed>,
        ui_fixed : nop_s<ui_out>;
 export ui_out : font_s<ui_in>;
}
```

file: ./open.both.unit

```
compound open.both {
 import ui_in : ui_s<ui_fixed>,
        ui_fixed : nop_s<ui_out>;
 export ui_out : both_s<ui_in>;
} {
 local c : open.color, f : open.font;
 link ui_fixed to c@ui_fixed,
      ui_fixed to f@ui_fixed, ui_in to c@ui_in,
      c@ui_out to f@ui_in, f@ui_out to ui_out;
}
```

Figure 16: An atom `open.font` that uses the open class pattern to add the font feature to a UI library, and a compound `open.both` that adds both the color and font feature to a UI library .

file: ./open.init.unit

```
atom open.init {
 import ui_fixed : nop_s<ui_out>;
 export ui_out : ui_s<ui_fixed>;
}
```

file: ./fixed_s.sig

```
signature fixed_s<ui_p> {
 class Widget extends Object
 { void setFont(int); void setColor(int);
   void paint(); }
 class Button extends ui_p.Button
 { void setLabel(String); }
 class Window extends ui_p.Window
 { void add(ui_p.Widget); void show(); }
}
```

file: ./open.fixed.unit

```
compound open.fixed {
 export ui_fixed : fixed_s<ui_fixed>;
} {
 local i : open.init, b : open.both;
 link i@ui_out to b@ui_in,
      b@ui_out to i@ui_fixed,
      b@ui_out to b@ui_fixed,
      b@ui_out to ui_fixed;
}
```

Figure 17: A compound `open.fixed` that fixes the features of a UI framework.

pound `open.both` itself uses the open class pattern so that it appears to directly add both the color and font features to a UI library.

Figure 17 shows how the open class pattern can be used end-to-end to create a UI library. The compound `open.fixed` uses the atom `open.init` and compound `open.both` to create a feature-complete UI library. The atom `open.init` provides the initial implementation of the UI library, so it does not need to import an in-package. Inside `open.fixed`, the features of the UI library are "fixed" by taking the out-package of unit instance *b* and connecting it to the fixed-package imports of both unit instances *i* and *b*. Figure 18 illustrates the linking done in both the compounds

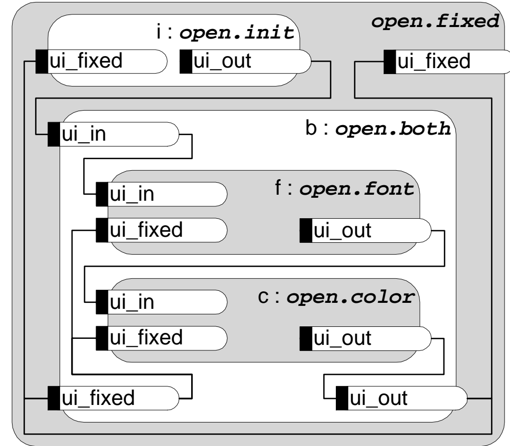

Figure 18: A graphical illustration of connections made in compounds `open.fixed` and `open.both`.

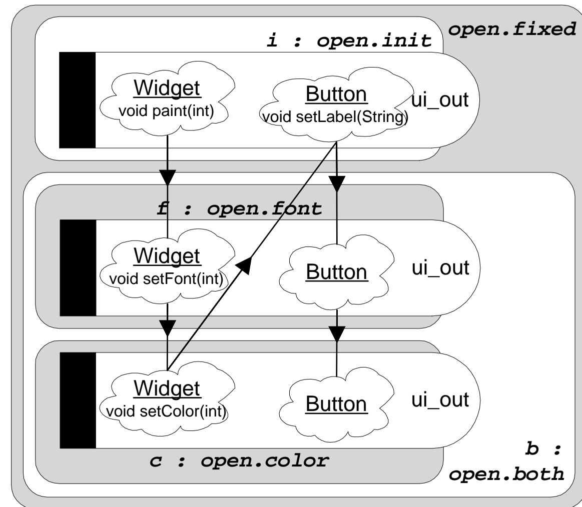

Figure 19: A graphical illustration of the global inheritance relationships established by `open.fixed`.

`open.fixed` and `open.both`.

No more features can be added to `open.fixed` using the open class pattern. The compound's exported package `ui_fixed` appears outside of the compound to be a UI library that provides the color and font features. All intermediate classes, those not exported from `open.fixed`, are hidden from clients of the UI library. As a result, clients of `open.fixed` are isolated from the fact that the UI library was built using the open class pattern.

Figure 19 shows the global class inheritance hierarchy established by the compound `open.fixed`. Classes in independently developed units can exist between each other in the class inheritance hierarchy! A color `Button` is both a subclass of the original `Button` and a subclass of a color `Widget`, which solves the extensibility problem in Figure 12.

When combined with cyclic component linking, mixins have a non-

trivial effect on type checking. *Inheritance cycles* could be introduced into the class hierarchy or *method collisions* could occur when two ambiguous methods exist in the same scope. As we show in Section 4, separate type checking in Jiazzi can disallow these constructions.

The open class pattern is not unique to Jiazzi; it can also be useful as a convention in Java code outside of Jiazzi components when separate compilation, and especially separate type checking, is not important and source code is open to modification. However, Jiazzi makes the open class pattern's use more realistic, and also enables configuration of features with external linking. The open class pattern, but not Jiazzi itself, necessarily causes a shift in the programming model of Java code that uses it to add features. At present, we are adding better support for the open class pattern in Jiazzi to minimize the effects of this shift.

4. TYPE CHECKING

The Java classes used in the construction of an atom are checked according to the Java Language Specification [12]. Jiazzi uses a conventional Java compiler to perform these checks; stubs are generated for imported classes to ensure that they are used correctly in classes that the atom contains. A Jiazzi component linker then ensures that the atom's classes are consistent with the atom's unit signature. For compounds, the linker must ensure that the linking of units within a compound is consistent with the compound's unit signature and the unit signature of units used to create unit instances in the compound. All of these checks are performed by type checking connections. Type checking each connection requires the matching of classes in a *source package* with classes in a *sink package*.

Classes contained in an atom are in potential source packages, while classes exported in an atom's unit signature are in sink packages. An atom's source packages are implicitly connected to the atom's sink packages by package name equivalence. Classes imported in a compound's unit signature and exported in the unit signatures of a compound's unit instances are in potential source packages, while classes exported in a compound's unit signature and imported in the unit signature of a compound's unit instances are in sink packages. A compound's source packages are explicitly connected to sink packages; source packages are on the left and sink packages are on the right of **`to`** clauses in a compound's **`link`** statements.

To compare the class signatures of sources and sinks, Jiazzi expands package signatures by replacing each package signature's package parameters with the names of the packages they are bound to. Expansion checks are made to ensure that a package parameter is bound to a package that provides all classes referred to through the package parameter. Package signatures have no other purpose during type checking other than being expanded to generate unit signatures.

The indirect properties of a class, such as subclasses, superclasses, and inherited methods, require that the class exist in an *environment* of other classes, over which references to classes in the signatures of these classes are closed. Properties for source classes are extracted from the source environment of a connection, and properties for sink classes are extracted from the sink environment of a connection. The source environment is the same for all connections in a unit, and is created using the union of the class signatures for all classes in potential sources and the class signatures of the unit's imported classes. For classes contained inside an atom, class signatures are extracted directly from their class definitions. The sink environment is created using the union of the class signatures for all classes imported and exported into the unit or unit instance where the sink package is located.

Type checking thus amounts to source–sink class matching in the context of a source environment and sink environment. Consider a connection from packages `source` to `sink` inside a unit u. If a class `C` is described in `sink`, then `C` must be described in `source`, otherwise u is rejected. In addition, the following rules must hold for u to be well-typed:

- R1. If method ***`m`*** is introduced in `sink.C`, then ***`m`*** must either be introduced in or an inherited method of `source.C` according to the source environment.
- R2. If `p.D` is a subclass of `sink.C` and a method ***`m`*** is introduced in `p.D` according to the sink environment, then ***`m`*** must not be introduced in nor be an inherited method of `source.C` according to the source environment.
- R3. If class `p.D` is the direct superclass of `sink.C` according to the sink environment, then each direct and indirect superclass of `source.C` is either the class `q.D` (where `q` is connected to `p`), a superclass of `q.D` according to the source environment, or a class that is not *visible* in the sink environment.

The first rule is straightforward; Rule R1 ensures that method requirements are met. A method provided to the sink class could be found in either the source class or one of its superclasses. Rule R2 rejects constructions that would cause method collisions. We have chosen an interpretation of method collision that disallows both silent overriding, where the signatures of colliding methods are the same, and ambiguous method calls, where the signatures of colliding methods differ only by return type.

Rule R3 ensures that superclass relationships are consistent between connections and it also prevents some subclassing relationships from being hidden. A class is not visible in the sink environment if is not exported from or imported into the unit that contains the sink. The rule ensures that subclassing relationships that are true locally within a unit are also true globally in correct unit compositions, while still allowing subclass relationship hiding to accommodate class hiding. The open class pattern in Section 3 relies on subclass relationship hiding, since intermediate classes in an open class construction are hidden to units that only import fixed classes.

Rule R2 depends on method scoping, in that it only checks for method collisions using methods visible in class signatures. We explain method scoping in more detail in Section 4.1. We discuss type checking in the presence of abstract methods in Section 4.2.

4.1 Method Scoping

A *method collision* occurs when two methods have conflicting types and are visible in the same class. Since not everything is visible in the source and sink environments used to check connections, then according to Rule R2, two methods can collide only if they are

file: .//icon/Icon.java
```
package icon;
public class Icon
 extends Object {
 public void paint() {
  ... draw(); ...
 }
 void draw() { ... }
}
```
file: .//cw/Cowboy.java
```
package cw;
public class Cowboy
 extends icon.Icon {
 public void duel() {
  ... draw(); ...
 }
 int draw() { ... }
}
```

Figure 20: Valid conventional Java source code that demonstrates Java's built-in package scoping.

visible in the same scope. This is important since, in order to reject method collisions modularly in the presence of mixins, method scope must be accounted for.

Without considering method scope, a method collision occurs in the conventional (non-Jiazzi) Java code of Figure 20. Even though the method ***void draw()*** already exists in its superclass `Icon`, the class `Cowboy` introduces the method ***int draw()***. However because of package scoping, this Java code is valid. Both ***draw*** methods are visible only in the enclosing package, because both lack public or protected access declarations. Since each class is in a different package, the scopes of the methods do not overlap and no ambiguity occurs. This observation is similar to the one made by Riecke and Stone [22] and elaborated on by Vouillon [28] with respect to class-based typing.

This same protocol is implemented in Jiazzi for unit scopes. Rule R2 does not consider methods that are hidden in class signatures. A method that is not mentioned in the class signature of an imported class is hidden from that unit; a method that is not mentioned in the class signature of an exported class is hidden outside of the unit. This hiding establishes method scopes, and if two methods do not exist in overlapping scopes, they cannot collide.

In some cases, method scope can be explicitly used to eliminate accidental method collisions through wrapping units in compounds. In Figure 21, the atom *mix.cowboy* exports a class `Cowboy` with both methods ***duel*** and ***draw***. A unit instance created using *mix.cowboy* is connected in the compound *cowboy.wrong*. Because the class `Icon` imported into the compound also contains the method ***draw***, a method collision occurs and *cowboy.wrong* is rejected.

Instead of rewriting the unit *mix.cowboy* to hide ***draw*** in `Cowboy`, a programmer can wrap the compound *hide.draw* around *mix.cowboy*, as shown in Figure 22. *hide.draw* hides the method ***draw*** from its public interface, which allows *hide.draw* to be used in *cowboy.right*.

file: ./cw_e_s.sig
```
signature cw_e_s<icon_p> {
 class Cowboy extends icon_p.Icon
 { void duel(); int draw(); }
}
```
file: ./icon_e_s.sig
```
signature icon_e_s<> {
 class Icon extends Object
 { void paint(); void draw(); }
}
```
file: ./cw_s.sig
```
signature cw_s<icon_p> {
 class Cowboy extends icon_p.Icon
 { void duel(); }
}
```
file: ./icon_s.sig
```
signature icon_s<> {
 class Icon extends Object
 { void paint(); }
}
```
file: ./mix.cowboy.unit
```
atom mix.cowboy {
 import icon_in : icon_s<>;
 export cw_out : cw_e_s<icon_in>;
}
```
file: ./cowboy.wrong.unit
```
compound cowboy.wrong {
 import icon_in : icon_e_s<>;
 export cw_out : cw_s<icon_in>;
} {
 local cw : mix.cowboy ;
 link icon_in to cw@icon_in,
      cw@cw_out to cw_out;
}
```

Figure 21: *cowboy.wrong* creates a method collision.

file: ./hide.draw.unit
```
compound hide.draw {
 import icon_in : icon_s<>;
 export cw_out : cw_s<icon_in>;
} {
 local cw : mix.cowboy;
 link icon_in to cw@icon_in,
      cw@cw_out to cw_out;
}
```
file: ./cowboy.right.unit
```
compound cowboy.right {
 import icon_in : icon_e_s<>;
 export run_out : run_s<>;
} {
 local cw : hide.draw ;
 link icon_in to cw@icon_in,
      cw@cw_out to cw_out;
}
```

Figure 22: *hide.draw* hides an unwanted method allowing the composition of a valid *cowboy.right*.

In some situations, a programmer would like to expose a pair of colliding methods to clients (e.g., both ***draw*** methods may need to be visible in `Cowboy`), and let the client programmer choose one. In Jiazzi, ambiguous methods that cannot be resolved using scope during composition are always rejected as method collisions. Moby [9], in contrast, allows ambiguous methods to be exposed in the same scope, and leaves the complexity of resolution to the caller.

4.2 Abstract Methods

In addition to instance methods and subclassing, other Java language features can be expressed in class signatures. Instance fields, static methods, and static fields are checked like instance methods. Constructors must be matched directly in the source class, because they are not inherited. Checking of abstract methods, however, deserves extra discussion.

Unlike concrete virtual methods, an abstract method within a class or interface cannot be hidden by a class signature. Otherwise, a non-abstract subclass of the class described by the class signature could contain hidden abstract methods. This restriction is present in MultiJava's open classes for the same reason [6].

Because of the need to upgrade libraries, Java allows abstract methods to be unimplemented in concrete classes [18]. Successive versions of Java core libraries have added abstract methods to existing classes (e.g., compare the initial and current version of class `java.awt.Graphics`). In Java, an abstract method invoked without an implementation will raise a runtime error. Jiazzi's requirement that concrete classes have no abstract methods conflicts with Java's binary compatibility support.

5. IMPLEMENTATION

Our current implementation of Jiazzi consists of a stub generator and an offline linker that operates on class files. The linker performs unit-level type checking, and it rewrites class files to form the binary forms of units. The binary forms of units can be used to create unit instances in a compound, or can be loaded into a Java Virtual Machine (JVM). Only the class file's constant pools, which contains its symbols, are rewritten by the linker: there is no need to parse and inspect the bytecode instructions in method bodies. Class file features such as debug attributes, which are important for compatibility with existing Java development tools, are preserved in the rewritten class files.

When compounds are linked, class file symbol rewriting is used to update references to imported classes in class files when connections are made in compounds, and to rename classes based on whether they are hidden or exported from the compound. Class file rewriting is also used to establish method scopes. Since method scopes are dependent on unit boundaries and not on class or package boundaries, we cannot depend on any built-in JVM mechanisms to delineate method scopes at runtime. Instead, class file rewriting renames hidden methods. So that no accidental collisions can occur in valid constructions, renaming is applied across multiple unit compositions so that distinct methods remain uniquely named.

The only run-time performance penalty due to using units arises from the duplication of the binary forms of units during linking. That is, using a single unit to create many different unit instances could lead to binary bloat, which can have negative performance effects (e.g., due to instruction cache and native compilation). On the other hand, units used in different contexts could be optimized independently. For example, method scoping could be used to devirtualize method calls [29] as units are linked.

Although the symbols used in Java class files can easily be rewritten, Symbols referred to in Java native methods cannot. A Java native method is bound to a method based on the name of the method and its containing class. Changing either the name of the class or method breaks this connection. Therefore, only classes without native methods can be contained within units. Such renaming also interferes with some uses of the Java Reflection API where symbols are referred to at runtime.

A linked unit can be loaded and linked directly by the JVM. Since such linking is primarily performed in the class loader, we refer to this as *class loader linking*, in contrast to Jiazzi linking, which has been described so far. The exported classes of the unit appear as normal Java classes, which can be loaded and be made available through the class loader.

Compared to Jiazzi linking, class loader linking is fragile. A class's imports can be bound to classes that differ from the classes compiled (and type checked) against. Since there is no description of the classes originally compiled against, like those provided by Jiazzi with a unit's signature, type checking during class loader linking is implemented in the JVM with incremental whole-program analysis (using constraints [17]) and runtime checks (e.g., checking that abstract methods are implemented when invoked).

Currently, component-based programs in Jiazzi must use a combination of Jiazzi and class loader linking. Many classes in the standard language library, such as **`Object`** and **`String`**, are strongly tied to the language and can only be linked through the class loader. Also, because they depend on reflection or native methods, many class libraries cannot be repackaged as Jiazzi components.

6. RELATED WORK

Many of the techniques and concepts used in Jiazzi have been explored previously: the core component model is derived from program units [7, 10] and Jiazzi's method-scoping rules resemble those of Riecke and Stone [22] and Vouillon [28]. Our contribution in Jiazzi is demonstrating how these techniques can be combined to define a practical component system for Java that also applies to other statically typed object-oriented languages. In doing so, we have solved the type challenge left open by Findler and Flatt [7].

The language ComponentJ [24] is a unit-like component system for Java. ComponentJ is a language extension in which components are objects that import and export methods but not types. Components in ComponentJ are also first-class values.

Moby [8, 9] is a structurally typed object-oriented language that supports ML-style modules. Methods can be hidden in modules: object types created in these modules do not propagate the hidden methods in their type. However, a hidden method can still be invoked by explicitly specifying its originating class type. Since Moby does not use subclassing relationships implicitly when typing method invocations, module applications that create method collisions are allowed. To resolve ambiguous methods, Moby relies

on object-view coercion to explicitly coerce the type of an expression from a class to one of its superclasses. Moby does not support the cyclic linking of modules.

Mixins were pioneered in CLOS [15]. JAM [2] extends Java with in-language mixins. The module system of Objective Caml [16] supports external class connections. Since classes can be defined in modules, these classes can also form something like mixins. However, Objective Caml does not permit a class supplied to a component (functor) to provide more methods than required by the component.

JavaMod [3] is a theoretical module calculus for Java. It supports the import and export of classes and cyclic module linking. However, they do not consider situations where imported classes inherit from exported classes. JavaMod supports subclassing across module boundaries, but extra methods provided for an imported class must be explicitly hidden, and resulting subclasses will not contain those hidden methods.

Jiazzi's open class pattern provides a modular way to add features to classes in object-oriented systems. Odersky [19] addresses the similar problem of class adaptability by adding views to objects. A view is an unnamed function that adds methods and fields to an existing class. Views cannot be implemented with separate compilation.

MultiJava [6] is a Java language extension that addresses adding new methods (but not fields) to existing classes with open classes. New methods can be added to a class using scoped compilation units. Separate compilation is supported since clients of the class explicitly choose which scopes they can view. It is possible for new compilation units in MultiJava to add new methods to classes after execution begins, in contrast to Jiazzi where new methods and fields can only be added when the units undergoes link-time construction.

Work in separation of concerns, such as subject-oriented programming [13], address the issues of separating class features into separate modules. Feature-oriented programming [21] and role components [27] use individual mixin-like structures to decompose designs into feature hierarchies. An approach similar to the open class pattern is used in Mixin Layers [25]. Instead of using individual mixins, Mixin Layers provides constructs that apply mixins to multiple classes at once. Java Layers [5] is an implementation of Mixin Layers for Java.

Jiazzi does not provide a solution for the configuration of runtime behavior as do other component systems such as COM [23], CORBA [20], and JavaBeans [14]. Such components are used at design time to configure runtime behavior and do not provide a good solution for system deployment. Configuration of code versus runtime behavior address reusability at different times and granularities. Jiazzi complements these component systems. For example, since a Bean in JavaBeans exists as a set of Java classes, it can be contained inside a Jiazzi unit.

7. CONCLUSIONS AND FUTURE WORK

We have presented the design and implementation of Jiazzi, which enhances Java with externally linked, hierarchical, and separately compiled components. Jiazzi's support for mixin constructions and cyclic linking allows open classes to be simulated leading to clean functional decomposition of features in programs. Jiazzi does not change existing Java development practice: programs are still written in the Java language and still execute on conventional Java virtual machines. Although we have finished Jiazzi's core component model and initial implementation, we are still enhancing Jiazzi in many areas:

- Providing more integrated support for open classes in Jiazzi;
- Adding more flexibility to component composition by providing more control over method scopes and the hiding of abstract methods;
- Add support for online linking of components, and a meta-programming protocol that allows for configuration of component linking at runtime instead of statically in the Jiazzi component language;
- Integrating Jiazzi more closely into a JVM so class loader linking and bytecode duplication can be avoided, and to allow reflection and native methods inside components.

We expect that more areas of improvement will be revealed as we gain experience in using Jiazzi to build large systems. An implementation of Jiazzi for Java is available for download at: ***http://www.cs.utah.edu/plt/jiazzi***.

Acknowledgments

We thank Don Batory, Richard Cardone, Craig Chambers, Eric Eide, Robby Findler, Kathleen Fisher, Alastair Reid, Patrick Tullmann, and the anonymous reviewers for comments on drafts of this paper. Sean McDirmid was supported in part and Wilson Hsieh in full by NSF CAREER award CCR–9876117. Matthew Flatt was supported by DARPA contract F33615-00-C-1696.

REFERENCES

[1] *Jiazzi Homepage*, 2001. http://www.cs.utah.edu/plt/jiazzi.

[2] D. Ancona, G. Lagorio, and E. Zucca. JAM – a smooth extension of Java with mixins. In *Proc. of ECOOP*, pages 154–178, June 2000.

[3] D. Ancona and E. Zucca. True modules for Java classes. In *In Proc. of ECOOP*, To appear.

[4] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. of OOPSLA*, pages 303–311, Oct. 1990.

[5] R. Cardone and C. Lin. Comparing frameworks and layered refinement. In *Proc. of ICSE*, pages 285–294, May 2001.

[6] C. Clifton, G. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proc. of OOPSLA*, pages 130–146, Oct. 2000.

[7] R. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proc. of ICFP*, pages 98–104, Sept. 1998.

[8] K. Fisher and J. Reppy. The design of a class mechanism for Moby. In *Proc. of PLDI*, pages 37–49, May 1999.

[9] K. Fisher and J. Reppy. Extending Moby with inheritance-based subtyping. In *Proc. of ECOOP*, pages 83–107, June 2000.

[10] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proc. of PLDI*, pages 236–248, May 1998.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, Reading, Mass., 2000.

[13] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proc. of OOPSLA*, pages 411–428, Nov. 1993.

[14] JavaSoft, Sun Microsystems, Inc. *JavaBeans Components API for Java*, 1997. JDK 1.1 documentation, http://java.sun.com/products/jdk/1.1/docs/guide/beans.

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

[16] X. Leroy, D. Doligez, J. Garrigue, D. R'emy, and J. Vouillon. The Objective CAML system, documentation and user's manual, 2000. http://caml.inria.fr/ocaml/htmlman/.

[17] S. Liang and G. Bracha. Dynamic class loading in the Java Virtual Machine. In *Proc. of OOPSLA*, Oct. 1998.

[18] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, Reading, Mass., 2000.

[19] M. Odersky. Objects + views = components? In *Proc. of Workshop on Abstract State Machines*, pages 50–68, Mar. 2000.

[20] OMG. *CORBA/IIOP Specification*, 2.4.1 edition, 2000. Formal document 2000-11-07. http://www.omg.org/-technology/documents/formal/corbaiiop.htm.

[21] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *Proc. of ECOOP*, pages 419–443, June 1997.

[22] J. G. Riecke and C. A. Stone. Privacy via subsumption. *Theory and Practice of Object Systems*, 1999.

[23] D. Rogerson. *Inside COM*. Microsoft Press, 1997.

[24] J. Seco and L. Caires. A basic model of typed components. In *Proc. of ECOOP*, pages 108–128, June 2000.

[25] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proc. of ECOOP*, pages 550–570, June 1998.

[26] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, N.Y., 1998.

[27] M. VanHilst and D. Notkin. Using role components to implement collarboration-based designs. In *Proc. of OOPSLA*, pages 359–369, Oct. 1996.

[28] J. Vouillon. Combining subsumption and binary methods: An object calculus with views. In *Proc. of POPL*, pages 290–303, Jan. 2001.

[29] A. Zaks, V. Feldman, and N. Aizikowitz. Sealed calls in Java packages. In *Proc. of OOPSLA*, pages 83–92, Oct. 2000.