Mixing COP and OOP

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

We describe and justify thirteen technical properties that a component system must possess to work with object-oriented programming languages (such as Java, C++, or C#). Our components are designed for large-scale, modular construction of programs with static checking of program compositions.

Jiazzi, our enhancement of Java, implements the technical properties that we describe. We use Jiazzi components in our examples, but only to make the discussion more concrete. Readers interested in the details of the Jiazzi component system should refer to our technical paper in OOPSLA '01 [8].

P1. **Language support**: Components should be described with a specific language construct.

P2. **Core language integration**: Components should contain, import and export instances of constructs in the core language.

As object-oriented software systems increase in size and complexity, components are becoming central to the design process, and they deserve language support. Otherwise, the lack of an explicit language construct for components places a substantial burden on programmers who implement components, and it obscures the programmer's intent to the compiler and other programmers.

Furthermore, components should be integrated with the core language, as opposed to working around the language with various compositional design patterns [6]. In class-based languages, the primary core language construct is the class, and large-scale elements of reuse are class libraries and frameworks.

Jiazzi's *units* [5] are explicit language constructs that describe components. There are two types of units: *atoms*, which are built from Java classes, and *compounds*, which are built from other units. Both atoms and compounds import and export Java classes.

P3. **Coarse-grained connections**: Connections between imports and exports should be able to connect many classes at once.

Connections between components should be coarse-grained, because components represent large-scale software entities. Components should import, export, and connect together groups of classes, since management of individual imported and exported classes does not scale with larger designs.

In Jiazzi, groups of classes are imported together, exported together, and connected together when units are linked; we call these groups of classes *packages* to emphasize their similarity to packages in standard Java.

P4. **Hierarchical composition**: A component should be composable into a larger, encapsulated component.

P5. **Component instantiation**: A component should be instantiated for each use.

P6. **External linking**: A component's external class dependencies should be resolved by the user of the component.

Composition of components to form larger, *compound* components enables the incremental construction of software. Outside a compound component, information about the components that were used in its construction should not be visible.

Component instantiation duplicates the structure of a component into a component instance. The instance then undergoes linking within a compound component. Thus, the distinction between "component" and "component instance" is similar to the distinction between "class" and "object." In particular, component instantiation allows a component to be used in multiple, independent contexts.

To support external linking, component interfaces should specify the shape of imports, but not the specific source of the imports. Specific connections among component instances should only be specified within a compound component. Avoiding hard-coded class dependencies among components makes the components as flexible as possible for client programmers [4].

Figure 1 provides a concrete example of unit composition. The atom *ui* exports package `ui_out`, which is a user interface (UI) library containing classes `Widget`, `Button`, and `Window`. The atom *applet* imports a package of UI library classes using the package `ui_in` and exports an applet with the class `Program` using the package `app_out`. The compound *linkui* links *ui* and *applet* together by creating the unit instances *u* and *a*, respectively. Using unit instances *u* and *a*, the exported package `ui_out` from *ui* is connected to the imported package `ui_in` in *applet*.

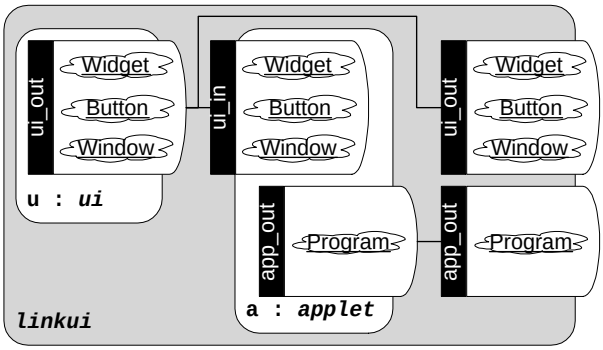

Figure 1: A graphical illustration of a compound *linkui* that composes two atoms *ui* and *applet*; methods are not shown.

P7. **Separate compilation**: Components should be compiled and type checked separately.

P8. **Signatures separated from implementation**: A component's class imports and exports should be defined in terms of signatures that are separate from class implementations.

Separate compilation enables development of large programs and deployment of components in binary form. The primary requirement of separate compilation is separate type checking [2]. Separate type checking, in turn, requires the use of signatures for classes, as opposed to the actual class implementations, when checking connections made between component imports and exports.

In Jiazzi, the signatures of classes in a unit's imported and exported packages are described using *package signatures*. A package signature can be used in multiple unit descriptions, which enhances the scalability of the Jiazzi component language. To support proper abstraction over signatures, package signatures have package parameters, which parameterize references to classes within the package signature.

Figure 2 shows example package signatures and units that use them. The package signature `ui_s` describes a UI library. The package signature `applet_s` describes an applet. The units *ui*, *applet*, and *linkui* from Figure 1 use these package signatures to describe their imported and exported packages. A package parameter called `ui_p` parameterizes both package signatures, and is bound to the appropriate UI library that is imported or exported into the unit.

Mixin Constructions

P9. **Implicit hiding**: A component should accept imported classes that supply more members (e.g., methods) than the component's import signature specifies; the extra members should not be visible inside the component.

P10. **Import subclassing**: A component should be allowed to define a subclass of an imported class.

Implicit hiding allows for more flexible composition by not requiring exact matches when connecting to a component's imports.

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(); }
}
```
file: ./applet.unit
```
atom applet {
 import ui_in : ui_s<ui_in>;
 export app_out : applet_s<ui_in>;
}
```
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 2: Package signature `ui_s` and `applet_s`, and the textual representation of atoms *applet* and *ui* and compound *linkui* from Figure 1.

Import subclassing, which implies inheritance across component boundaries, is necessary for grouping classes and class extensions into components. In particular, combining implicit hiding with import subclassing enables the composition of class-extending components; e.g., *mixins* [1].

Figure 3 illustrates how mixins work in Jiazzi using an example compound *open.fixed* that composes three atoms: *open.init* that provides a basic UI library, and *open.font* and *open.color* which respectively add font and color features to a UI library using mixin constructions. In the illustration, we track only the construction of the class `Widget`, and we do not show all imports and connections.

In the compound *open.fixed*, the package `ui_out` exported from an instance of *open.font* is connected to the package `ui_in` imported into an instance of *open.color*. The method ***setFont***, provided by the export of `Widget` in *open.font*, is not visible in *open.color*. An important feature of implicit hiding is that it is local to the inside of a component. In particular, because `ui_out.Widget` subclasses `ui_in.Widget`, the export of `Widget` from the instance of *open.color* does contain ***setFont***. Therefore, `Widget` can be exported from the compound with both ***setFont*** and ***setColor*** methods.

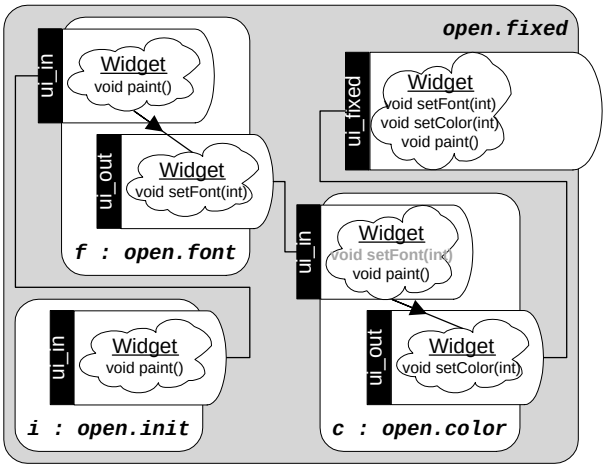

Figure 3: A graphical illustration of a compound that uses a mixin construction; inheritance relationships are directed arrows pointing to subclasses; methods visible in export/import signatures are listed in classes; gray methods are not visible.

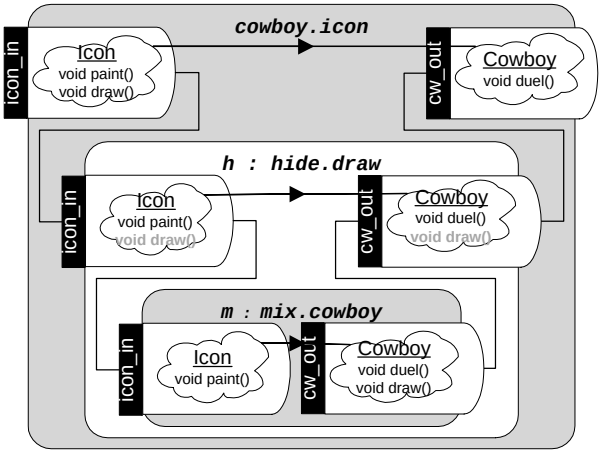

Figure 4: A graphical illustration of a composition where method scope is used to avoid method collisions; gray methods are not visible.

Supporting mixin constructions creates the possibility of unresolvable ambiguous methods, known as *method collisions*. For example, linking two instances of `open.font` together should lead to an error, since the unit instances add the same method. To detect method collisions in the presence of separate type checking (Property P7), a method's scope must be defined with respect to component boundaries. Methods that are not visible in the same component scope are not ambiguous, and therefore should not cause method collisions.

Figure 4 shows an example where unit composition depends on method scope to avoid method collisions. The atom `mix.cowboy` exports a class `Cowboy` with methods ***`duel`*** and ***`draw`***, as in "draw your guns." The compound `cowboy.icon` imports a class `Icon` with method ***`paint`*** and ***`draw`***, as in "draw an icon." Linking `mix.cowboy` directly in `cowboy.icon` is not possible; the ***`draw`*** methods in `Icon` and `Cowboy` are distinct so cannot exist in the same class in the same scope. To ensure the ***`draw`*** methods do not appear in the same scope, the compound `hide.draw` hides the ***`draw`*** method in `Icon` on import and the ***`draw`*** method in `Cowboy` on export.

P11. **Explicit hiding**: A component should be allowed to export class implementations that supply more members (e.g., methods) than the component's export signature specifies; the extra members should not be visible outside the component.

P12. **Export subclassing**: A component should be allowed to import a subclass of an exported class.

P13. **Cyclic component linking**: Component linking should resolve mutually recursive dependencies among components.

Explicit hiding enhances component reuse, because irrelevant methods and fields can be hidden by wrapping a component. Export subclassing combined with explicit hiding enables the formation of "upside-down" mixins, which behave the same as normal mixins, just from a different perspective. We explain how such upside-down mixins are useful in the following section.

Cyclic component linking enables natural component organizations, because mutually recursive "has a" relationships are especially common at the class level, and can naturally span component boundaries. "Upside-down" mixins necessarily require cyclic component linking.

Combining cyclic component linking with mixin constructions creates the possibility that inheritance cycles could be introduced into the class hierarchy. To detect inheritance cycles in the presence of separate type checking (Property P7), the actual subclassing relationship for two classes should always be locally obvious when they are visible in the same component scope. Thus, components must not be allowed to hide subclass relationships when the classes involved are visible.

Open Class Pattern

Mixins and "upside-down" mixins can be combined in the *open class pattern*, which is used to 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 [3]. With the open class pattern, we can replace the use of many design patterns [6] 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.

Figure 5 illustrates how the open class pattern works using an example in Jiazzi of an atom `open.color`, which was partially described in Figure 3, that adds the color feature to a UI library. The unit imports the previous implementation of the classes in the package `ui_in`. These classes are subclassed by exported classes in the package `ui_out`, forming normal mixins. The key to the open class pattern is that the unit must also import the fixed UI library, whose classes subclass classes in `ui_out`, which uses upside-down mixins.

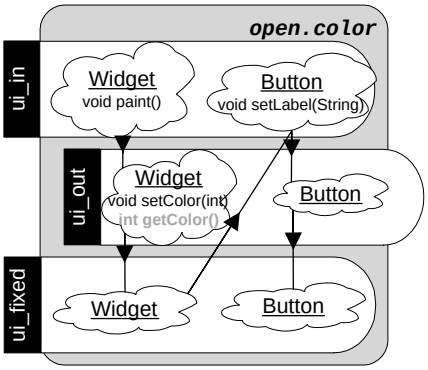

Figure 5: A graphical illustration of the subclassing relationships and methods of `Widget` and `Button` local to an atom *`open.color`* that adds the feature of color to a UI library; gray methods are explicitly hidden.

Using the open class pattern ensures that the class `Button` is always a subclass of `Widget`, no matter how many feature-adding mixins are applied to `Widget`. In this illustration, the method ***`getColor`*** in class `ui_out.Widget` is not visible outside of *`open.color`* because it is explicitly hidden. An important feature of upside-down mixins is that the subclassing relationship is visible, so inherited methods that are not visible outside a component can be visible inside the component. In particular, due to `ui_fixed.Widget`'s upside-down mixin relationship with `ui_out.Widget`, and `ui_out.Button`'s normal mixin relationship with `ui_fixed.Widget`, the method ***`getColor`*** is visible in both widgets and buttons.

Figure 6 illustrates how the open class pattern is used in an end-to-end construction using the compound *`open.fixed`*, which was partially described in Figure 3. The atom *`open.init`* provides the initial implementation of a UI library. The atom *`open.font`* is implemented like *`open.color`* to add the font feature to a UI library. The compound *`open.fixed`* links these units together into a single UI library with both the color and font features. The need for cyclic linking is apparent when we must "fix" the UI library's features by linking the exported package `ui_out` from unit instance *`c`* back to imports in the other unit instances.

Discussion

Seco and Caires [9] define a properties for components that differs significantly from ours. They argue that classes and inheritance, while usable in component implementations, should not be used across component boundaries. We believe that although inheritance is semantically fragile, it is well understood and easier to use than alternatives such as object aggregation. Our full OOPSLA paper [8] contains a more complete discussion of related work.

One necessary property missing from our list is online linking (i.e., dynamic linking), where components should be composable under program control after program execution has begun. The properties on our list do not interfere with online linking; some properties, such as external linking and separate compilation, facilitate online linking. Part of our future work is to provide support for online linking in Jiazzi.

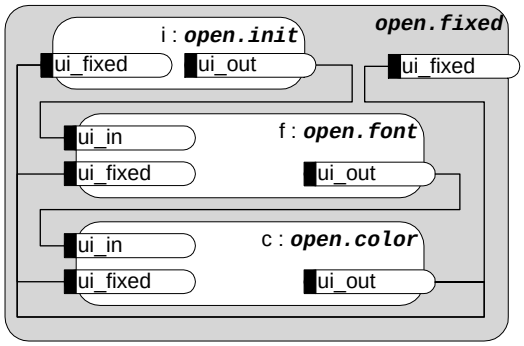

Figure 6: A graphical illustration of a compound that combines the atoms *`open.init`*, *`open.font`*, and *`open.color`* into a complete UI library with the features of both color and font; classes are not shown.

We have presented thirteen properties that are necessary in a component system to adequately support COP in OOP languages. These properties enable expressive component compositions. When combined, they also create challenges that must be overcome. For example, support for mixin constructions and cyclic linking supports the clean functional decomposition of features in programs, but also creates problems of method collisions and inheritance cycles. However, as we show in our work with Jiazzi, these challenges can be overcome.

REFERENCES

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

[2] L. Cardelli. Program fragments, linking and modularization. In *Proc. of POPL*, pages 266–277, Jan. 1997.

[3] 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.

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

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

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

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

[8] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *To Appear in the Proc. of OOPSLA*, Oct. 2001.

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