Java Component Development in Jiazzi

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

Extended Abstract

Current Java constructs for code reuse, including classes, are insufficient for organizing programs in terms of reusable software components. Although packages, class loaders, and various design patterns 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.

Jiazzi [4] enhances Java with support for large-scale software components. Jiazzi components are constructed as *units* [3]. A unit is conceptually a container of compiled Java classes with support for "typed" connections. There are two types of units: *atoms*, which are built from Java classes, 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 definitions, they enhance the component language's scaling properties.

Figure 1 illustrates how Jiazzi can be used in a simple component-based design where an application component is linked with a component that provides a user interface (UI) library. The atom `ui` exports UI library classes in package `ui_out`, while the atom `applet` imports UI library classes in package `ui_in` and exports an application class in package `app_out`. The compound `linkui` links `ui` and `applet` together into a complete program; connections between units are shown as lines between packages.

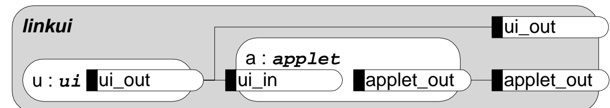

Figure 1: A graphical illustration of a compound `linkui` that composes two atoms `ui` and `applet`.

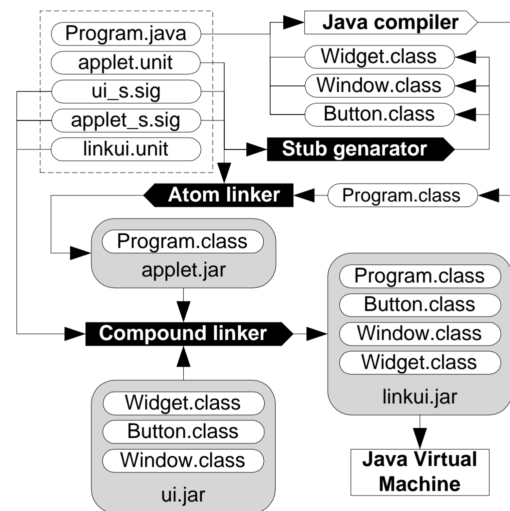

Figure 2: The files and development process of building the atom `applet` and the compound `linkui`.

The process of developing the design in Figure 1 is illustrated in Figure 2. The atom `applet` is defined in the file `applet.unit`. The package `ui_in` imported into `applet` is described using the package signature `ui_s` and consists of Java classes `Widget`, `Button`, and `Window`. The package `app_out` exported from `applet` is described using the package signature `applet_s` and consists of a single Java class `Program`, which is provided by a conventional Java source file `Program.java`. The package signatures `ui_s` and `applet_s` are respectively defined in files `ui_s.sig` and `applet_s.sig`.

The Java source file `Program.java` is shown in Figure 3. No specific provider of atom `applet`'s imported package `ui_in` is hard coded. Instead a *stub generator* is used to generate stub class files based on the package signature `ui_s`. This allows `Program.java` to be compiled using a normal Java source compiler

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 void run() { show(); }
 public  Program() {
  b.setLabel("start"); add(b);
 }
}
```

Figure 3: The contents of the Java source file `Program.java`.

(e.g., JDK's javac). An *atom linker* performs type checking to ensure that the result of the source compilation, `Program.class`, conforms to the structure of `Program` specified in package signature `applet_s`. After type checking, `Program.class` is packaged into the archive file `applet.jar`.

The compound *`linkui`* links the atom *`applet`* with a UI library provided by the atom *`ui`*, which is constructed into the archive file `ui.jar` in a manner similar to *`applet`*. A *compound linker* performs connection-oriented type checking and duplicates the class files of each atom. The duplicated class files are rewritten according to how connections are made in the compound; for example references to classes imported in package `ui_in` in *`applet`*'s `Program.class` are replaced with classes exported in package `ui_out` obtained from *`ui`*. Finally, the duplicated class files are packaged into the archive file `linkui.jar` by the compound linker. Since the compound *`linkui`* has no imports, its classes can safely be executed in a conventional Java Virtual Machine, for example by placing `linkui.jar` in the classpath.

Jiazzi supports advanced component programming in Java with the following features:

- **External linking:** external class dependencies of a component are resolved by the user of the component, even if the component is in binary form. Avoiding hard-coded class dependencies among components makes the components as flexible as possible for client programmers [2].
- **Hierarchical composition:** multiple components can be combined into a larger, encapsulated component that is not necessarily a self-contained program. Hierarchical composition allows for the incremental construction of software.
- **Separate compilation:** a component's source code can be compiled (type checked) independently of other components. The use of a component at link time can be compiled (type checked) without knowledge of its hidden implementation. Separate compilation enables development of large programs and deployment of components in binary form.
- **Flexible hiding:** a component can accept imported class implementations that supply more methods than it expects; a component can also export class implementations that supply more methods than its clients expect. Such hiding allows for flexible composition by not requiring exact matches when connecting to a component's imports, and allows for flexible encapsulation by allowing a component to restrict access to exported classes.
- **Inter-component subclassing:** a component can define a subclass of an imported class; a component can also import subclasses of its exported classes. Inter-component subclassing is necessary for grouping classes and class extensions into components.
- **Cyclic linking:** component linking can resolve mutually recursive dependencies among components. Cyclic linking enables natural component organizations, because mutually recursive "has a" relationships are especially common at the class level, and can naturally span component boundaries.

Jiazzi supports the composition of class-extending components; e.g., *mixins* [1]. Mixins and cyclic component linking can be combined into an *open class pattern*, which allows independent features that cross cut class boundaries to be packaged in separate components. 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.

Separate type checking is an essential part of separate compilation, and is often at tension with other Jiazzi features. By allowing mixins and cyclic component linking, we must disallow constructions of cycles in the class hierarchies or distinct methods that are ambiguous in a class. Using a novel "truth in subclassing" unit-level type checking rule and by enforcing method scope at component boundaries, Jiazzi is able to avoid or detect/reject such constructions while still type checking components separately.

Jiazzi does not change current Java programming practices; Jiazzi requires neither extensions to the Java language nor special conventions for writing Java code that will go inside a component. Java developers continue to use existing tools, including Java source compilers and IDEs, in the development of Java code. Component boundaries are not restricted enabling Java classes to be grouped into components where natural, which also makes it easier to retrofit legacy Java code into component-based designs.

We encourage Java developers to use Jiazzi. More information about Jiazzi can be found in our technical paper in OOPSLA '01 [4], or our web site, where an implementation is also available:
`http://www.cs.utah.edu/plt/jiazzi`

REFERENCES

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

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

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

[4] 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.