Jiazzi User Manual And Tutorial
Version 2.0

Sean McDirmid
School of Computing
University of Utah

http://www.cs.utah.edu/plt/jiazzi
jiazzi-users@flux.cs.utah.edu

February 20, 2002

Contents

Chapter 1

Introduction

Jiazzi is a component definition and linking language that can be used on code written in the unextended Java language. Java supports separate compilation, external linking, hierarchical structure, cyclic component dependencies, and class inheritance across component boundaries. Jiazzi processes conventional Java class files allowing the use of normal Java tools to develop and execute Java code.

The goal of this document is to describe the things you need to know to start using Jiazzi, including:

- An overview of Jiazzi's features (Chapter 2).
- How to install Jiazzi on your system (Chapter 3).
- A tutorial that goes through an example application written using Jiazzi's features (Chapter 4).
- A brief technical overview of Jiazzi (Chapter 5), including a discussion of syntax (Section 5.2) and tools (Section 5.5). Definitions in this section are hyper linked throughout the manual.
- A trouble shooting section to get solutions to common problems (Chapter 6).

A complete description of Jiazzi will be found in a language report. Also, reading the Jiazzi OOPSLA paper [5] will help, but please note it uses a different syntax and does not present all of Jiazzi's features.

Chapter 2

Overview

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 [5] 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.

Jiazzi supports advanced component programming in Java with the following basic 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.

Chapter 3

Installation

Before installing Jiazzi, a recent version of the Java Development Kit (JDK) should be installed that supports Java 2. Jiazzi is pure Java and can be installed on both Unix and Windows platforms. Jiazzi uses ANTLR for parsing that can be downloaded from `http://www.jguru.com`. For convenience, we have included the ANTLR runtime in the Jiazzi distribution directory.

The Jiazzi distribution can be downloaded from `http://www.cs.utah.edu/plt/jiazzi` as a GNU zip (`.gzip`) archive. These archives can be uncompressed by WinZip (on Windows) or gunzip (on unix). The distribution extracts the following files a directory called `jiazzi/`, which we will refer to as the Jiazzi distribution directory:

`jiazzi.jar`	A Java archive (JAR) file that contains the Java classes for all Jiazzi tools.
`antlr.jar`	The ANTLR runtime included in the distribution for convenience.
`validation/*`	Files used for installation validation.
`tutorial/*`	Files in this directory are used in the tutorial.

Once extracted, ensure that both `jiazzi.jar` and `antlr.jar` are in your CLASSPATH when executing Jiazzi tools. This can be done by either adding the files to the CLASSPATH environment variable or by adding them to the CLASSPATH when Java is invoked.

Note: If you are using a DOS shells, replace all forwardslashes with backslashes in file paths for the purposes of using this manual. Jiazzi, like Java, adheres to File.separatorChar which changes depending on the shell you use.

The files in the `validation/` directory are there for ensuring that Jiazzi is installed correctly. To test the installation, execute the linker while in the Jiazzi distribution directory (with `jiazzi.jar` and `antlr.jar` in your CLASSPATH) to test your installation:

```
java -cp validation/test.jar test.Test
mv validation/test.jar validation/test.old.jar
java jiazzi.Main validation/ -link test
java -cp validation/test.jar test.Test
```

Each type the class `test.Test` is executed, “Hello World” should be printed.

Chapter 4

Tutorial

The core component construct in Jiazzi is known as a *unit*. A unit is conceptually a container of compiled Java code with an explicit description of this compiled Java code's imports and exports. There are two types of units: *atoms*, which are built directly from Java class files (the compiled form of Java classes and interfaces), and *compounds*, which are built from atoms and other compounds.

Units import and export Java classes. Classes imported into a unit must be 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. To Java code, Jiazzi packages are realized as actual Java packages.

All units have a *unit signature*, which describes the structure of a unit's imported and exported packages. Packages in unit signatures can be described using *package signatures*, which consists of the shapes for a group of classes. Package signatures can be reused in multiple unit signatures. Package signatures can also undergo composition with other package signatures.

In this tutorial, we will go through the creation of a simple maze game [4] using units in Jiazzi. In doing this tutorial, one can get an understanding of not only how to use Jiazzi, but what to use Jiazzi for. All source files and some pre-linked units used for this tutorial are located in the `tutorial/` in Jiazzi distribution directory.

The rest of this section is organized as follows:

- The organization of signatures are presented first in Section 4.1. Package signature are described in Section 4.1.1. Package signatures are used in the construction of unit signatures, which are are described in Section 4.1.2.
- Unit definitions are presented next in Section 4.2. The definition of atoms are described in Section 4.2.1. The definitions of compounds are described in Section 4.2.2.
- The open class pattern, presented in Section 4.3, is a useful pattern in Jiazzi that enables the extensibility of classes without editing their source code. Two mechanisms in Jiazzi make the open class pattern easier to apply. First, package signature composition is described Section 4.3.1. Next, package inheritance is described in Section 4.3.2.

- Jiazzi supports variables, which are presented in Section 4.4, to enhances unit reusability. Lexemes, described in Section 4.4.1, enable parameterization of the names of classes and class members while gnerics, described in Section 4.4.2 enable parameterization with non-inherited or instantiated classes within a unit.

4.1 Signatures

Signatures in Jiazzi specify the structure of classes, packages, and units, which is used to ensure correct unit composition. Signatures are separated from definitions for the same reuse reasons that separating interfaces from class implementations is encouraged in normal Java.

There are three types of signature. First, a class signature specifies the methods, fields, constructors, and superclasses of an individual class. Second, a package signature is a collection of class signatures that describes the structure of a package of classes. Finally, a unit signature is a collection of package signatures that describe the structure of a unit's imported and exported packages.

Class signatures cannot be specified outside of a package signature. As such, only package and unit signatures are constructs in Jiazzi that can be manipulated independently. Package signatures exist in their own files and are parameterized to ensure maximal reuse in unit signatures. Unit signatures are a part of definition files. The following two sections describe package signatures and unit signatures.

4.1.1 Package Signatures

Package signatures are constructs that are used to describe the visible structure of classes in Java packages. They consist of a series of class signatures that each resembles a Java source class definition without method bodies or field initializers. Here is an example of a simple package signature:

```
signature maze.base = {
  package maze, game, io;
  class Room extends Object {
    Room();
    maze.Door getDoor(int dir); print(io.PrintStream out);
    ... /* most of Room's class signature is not shown */
  }
  class Door extends Object {
    Door();
    boolean canPass(game.Actor actor);
  }
}
```

The above package signature describes a maze with two constructs: a `Room` class that describes rooms in an adventure game, and a `Door` class that connects rooms together. The `Door` class has

a method `canPass` that determines whether or not actors in the adventure game can pass through the door.

Class signatures are parameterized by the enclosing package signature's *package parameters*. Package parameters are bound to packages (either imported, exported, or accessed through the CLASSPATH) when a package signature is used to describe a package in a unit. The package parameters are declared in a package signature using the **package** keyword. The package signature *`maze.base`* declares three package parameters: `maze`, `game`, and `io`.

Except for `Object` and `String`, classes can only be referenced through package parameters within a package signature. For example the class `Room` described in package signature `maze.base` has a method `getDoor` that returns an instance of `maze.Door`. Package self reference cannot be established by a package signature; e.g., a package signature cannot refer to the class `Door` described by the package signature `maze.base` directly.

Class signatures are constructed like class definitions in Java source files. Members of a class are defined within the class signature, which include constructors, methods, and fields. No initializers or method bodies can be expressed by a class signature, and instead member declarations are always proceeded by a semicolon. Constructors can be identified by the name of the class followed by the constructor argument list (e.g., `Door();`) or just by the argument list (e.g., `();`) as constructors in Java are unnamed.

A strong effort has been made to preserve the Java language in class signatures. Class signatures can be nested in other class signatures to declare inner class members. The modifiers **public**, **protected**, **final**, **abstract**, and **static** are supported for classes and class members when appropriate. The use of the **public** modifier is redundant as classes and class members are always public by default. The **private** and package-only access modifiers can never be used in a package signature since signatures only express intra-package interfaces.

Package signatures do not have meaningful identity. Packages that are described using the same package signature have no intrinsic relationship. A package signature must be located in a file that has the same name as the package signature and has the file extension `.sig`. This file must be located in the project path. For example, the file `maze.base.sig` is located in the `jiazzi/tutorial/` directory, which is the project path for this tutorial.

4.1.2 Unit Signatures

A unit signature describes the structure of a unit's imported and exported packages, each of which is described using a package signature. The package parameters of a package signature used to describe a package in a unit must be bound either to packages that are also imported or exported into a unit, or to packages that are accessible through the CLASSPATH. Consider the following atom *`edu.utah.maze.driver.one`*:

```
signature game.base = {
  package game, maze;
  ...
}
signature program = {
```

```
  class Main extends Object {
    static void main(String args[]);
  }
}
atom edu.utah.maze.driver.one {
  import maze : maze.base;
  import game : game.base;
  export out : program;
  bind package
    maze to maze@maze, maze to game@maze,
    game to game@game, game to maze@game,
    java.io to maze@io;
}
```

The atom *edu.utah.maze.driver.one* uses package signatures *maze.base*, *game.base*, and *program*. The package signature *game.base* has two package parameters `game` and `base`. Packages are bound with bind package clauses to package parameters in a unit and have the following syntax:

`source to destination@parameter`

Where `source` is an imported or exported package of the unit, or a package loaded through the CLASSPATH, `destination` is an imported or exported package of the unit, and `parameter` is a package parameter of the package signature used to describe `destination`.

Package parameters of package signatures used to describe packages in a unit must be bound from either the imported and exported packages of the unit, or packages that are loaded through the CLASSPATH. In the atom *edu.utah.maze.driver.one*, the package parameter `game` of the packages `maze` and `game` are both bound from the imported package `game`. The package parameter `io` of the package `maze` is bound from the package `java.io`, which is loaded through the CLASSPATH.

The result of package parameter binding is to replace references to classes through package parameters to references to classes through packages that are visible in the unit. Thus package signatures can be used in many units since the bindings of their package parameters is customizable.

The destination package of a bind package clause statement can be "wild carded" by specifying a star symbol (*). Wild cards match all imported and exported packages in a unit that have the package parameter specified. Wild carding is very useful in large units where package parameters with the same name are always bound to the same package. For example, the atom *edu.utah.maze.driver.one*'s bind package clauses can be rewritten as follows:

```
atom edu.utah.maze.driver.one {
  ...
  bind package
    maze to *@maze, game to *@game,
    java.io to maze@io;
}
```

A unit signature must be located in a unit definition file. This has the same name as the unit, has the file extension .unit, and must be located in the project path.

4.2 Unit Definitions

There are two kinds of units definitions. First, atoms (described in Section 4.2.1) are units that are linked directly out of Java class files. The unit definition of an atom consists of its unit signature and a set of class files that are used in the construction of the atom's linked form. Second, compounds (described in Section 4.2.2) are units that link other units. The unit definition of a compound consists of its unit signature, a linkage section, and a set of linked units that are used in the construction of the compound's linked form.

4.2.1 Atoms

The contents of an unit definition file (its .unit file) for an atom only consists of the unit signature that describes it. An atom is defined by Java code that is used to create classes provided by the atom. This is done by first running the stub generator if necessary, compiling the Java source code into class files using a Java source compiler, and then using the linker to create the atom's linked form.

For the purposes of this tutorial, we have already provided the directory jiazzi/tutorial/edu.utah.maze.driver.one/ that contains the Java source file implementation of the atom *edu.utah.maze.driver.one*. Java source code for an exported class goes in the following file:

<projectpath>/<atomname>/<exportedpackage>/<classname>.java

Where project path is described in Chapter 5, atom name is the name of the atom with the exported class, and exported package is the package that the class is exported under. As in conventional Java, if the exported package has a hierarchical name, then there is a subdirectory for each of the package's parents.

Java source files contain conventional Java source code. The Java source code for the class out.Main exported from the atom *edu.utah.maze.driver.one* is located in the file tutorial/edu.utah.maze.driver.one/out/Main.java which contains the source code like this:

```
package out;
public class Main extends Object {
  public static void main(String[] args) {
    ... /* normal Java method body not shown */
  }
}
```

Stubs for imported classes are needed to compile an atom's Java source files using a normal Java compiler. Stubs are created using the stub generator, which assuming one is in the directory jiazzi/ the following command must be run:

```
java jiazzi.Main tutorial/ -stub edu.utah.maze.driver.one
```

After stubs are generated, the Java source must be compiled using a standard Java source compiler, e.g., `javac` in the JDK. The Java source of the atom *edu.utah.maze.driver.one* can be compiled with the following command from the `jiazzi/` directory:

```
cd tutorial/edu.utah.maze.driver.one && javac -cp ./stubs.jar */*.java && cd ../../
```

After Java source has been compiled into Java class files, the atom's linked form can be generated using the linker. Again from the `jiazzi/` directory:

```
java jiazzi.Main tutorial/ -link edu.utah.maze.driver.one
```

The result is the JAR file `tutorial/edu.utah.maze.driver.one.jar` that is the atom *edu.utah.maze.driver.one*'s linked form.

4.2.2 Compounds

The unit signature of a compound is like that of an atom except that the first keyword is **compound** rather than **atom**. In a compound's unit definition file, the unit signature is followed by a linkage section, which completely defines the compound. Consider the following compound *edu.utah.maze.game.one*:

```
compound edu.utah.maze.game.one {
  export main : program;
} {
  local core : edu.utah.maze.core, main : edu.utah.maze.driver.one;
  link package
    core@outmaze to main@maze, core@maze,
    core@outgame to main@game, core@game,
    main@out to main;
}
```

The unit signature of a compound is specified in the first set of curly braces ({}). The second set of curly braces is the compound's linkage section. There are two kinds of basic clauses in the linkage section: a clauses that declare local units and link package clauses.

Local clauses instantiate (duplicates) units and assigns names to instantiations that are local to the containing compound. We refer to an instantiated unit as a local unit. In the compound *edu.utah.maze.game.one*, the atoms *edu.utah.maze.core* and *edu.utah.maze.driver.one* are each instantiated once creating the local units `core` and `main`. A local unit is a duplicate of the unit that is instantiated to create it, and local units that are instantiated from the unit have there own unique imported and exported packages that correspond to the imported and exported packages.

Link package clauses make connections between the imported and exported packages of a compound's local units and its own imports and exports. Each statement in link package clause has one of the following form:

- `local-source@export-pkg` **to** `local-destination@import-pkg`
- `compound-import-pkg` **to** `local-destination@import-pkg`
- `local-source@export-pkg` **to** `compound-export-pkg`

In the compound *edu.utah.maze.game.one*, the first statement of the link package clause links the package `outmaze` exported from the local unit `core` to the package `maze` imported into the local unit `main`. Link package clauses can connect packages imported into a compound to packages imported into a local unit, and they must connect packages exported from local units to packages exported from the compound. For example, the last statement of the link package clause links the package `out` exported from the local unit `main` to the package `main` exported from the compound.

Like statements in bind package clauses, statements in link package clauses can use wild cards (*) in the local unit destination position. A wild card matches a local unit that imports and the compound that exports a package with the same name as the sink package. The compound *edu.utah.maze.game.one*'s package link clause can be rewritten as follows using wild cards:

```
compound link.maze.game.one {...} {
  ...
  link package
    core@outmaze to *@maze, core@outgame to *@game,
    main@out to out;
}
```

We have not shown you the source for the atom *edu.utah.maze.core* yet. However, it is prelinked in the tutorial directory (`edu.utah.maze.core.jar`) so the compound *edu.utah.maze.game.one* can be linked. The compound can be linked by the linker, using the following command from the directory `jiazzi/`:

```
java jiazzi.Main tutorial/ -link edu.utah.maze.game.one
```

The result is the JAR file `jiazzi/tutorial/edu.utah.maze.game.one.jar` that is the compound's linked form. To ensure that the compound is linked correctly, the main method provided by the driver can be executed with the following command:

```
java -classpath tutorial/edu.utah.maze.game.one.jar main.Main
```

If it works, congratulations! You have just created your first program using units. If it does not work, something may be setup wrong. Hopefully the answer to your problems lies in Chapter 6, otherwise contact the author (`jiazzi-users@flux.cs.utah.edu`).

4.3 Open Class Pattern

So far, this overview has talked about basic package import and export features of Jiazzi. Jiazzi also allows subclassing across unit boundaries (e.g., deriving a subclass of an imported class) and

supports basic class extending components (mixins [1]). Class imports, exports, and subclassing between units can be combined into an *open class pattern*, which allows the addition of features to classes without changing the subtyping relationships between the classes or modifying their source code. In the technical paper [5], the mechanics and purpose of the open class pattern are discussed in depth. The purpose of this section in the tutorial is to introduce the usefulness

Two mechanisms in Jiazzi make the open class pattern easier to apply. The first mechanism is *package signature composition*, which is described in Section 4.3.1. Package signature composition enables new package signatures to add to the descriptions of old package signatures. The second mechanism is *package inheritance*, which is described in Section 4.3.2. Using package inheritance, blanket class extensions can be specified between packages imported and exported in a unit.

4.3.1 Package Signature Composition

Package signature composition allows a new package signature to be defined using existing package signatures. The result is that new classes and class members can be added in addition to what the composed package signatures already describe. Consider the following package signature *maze.securable* that composes the package signature *maze.base*:

```
signature maze.securable = z : maze.base + {
  package maze, game, io;
  bind package maze to z@maze, game to z@game, io to z@io;
  class Door {
    boolean canOpen(game.Player p);
  }
}
```

Composed package signatures must be assigned identifiers within the composing package signature so their package parameters can be bound. In the package siganture *maze.securable*, the package signature *maze.base* is composed and assigned the composed signature identifier `z`.

Package parameters are bound through a bind package clause. Source package parameters are the package parameters of the composing package signature and are located on the left of the **to** statements. Sink package parameters are those of composed package signatures, which are separated from the composed package signatures using the @ symbol, and are located on the right of the **to** statements. In the package signature *maze.securable*, the package parameter `maze` is bound to the package parameter `maze` of the composed package signature *maze.base* as identified by `z`.

The result of package signature composition is a package signature that reuses and can add to the composed package signature. For example the package signature *maze.securable* describes the class `Door` with the methods `canPass`, which was described in the composed package signature, and `canOpen`, which is added by the package signature *maze.securable*. As we will see later in Section 4.4.1, package signature composition can be used to combine multiple previously defined package signatures.

Bind package clauses are also used in unit signatures to bind package parameters, except rather than binding package parameters from other package parameters, package parameters in units are

bound from actual packages. As in unit signatures, bind package clauses can be used with wild cards.

4.3.2 Package Inheritance

Consider the following atom:

```
atom edu.utah.maze.securable {
  import inmaze : maze.base;
  export outmaze extends inmaze : maze.securable;
  import maze extends outmaze;
  import game : game.base;
  bind package
    maze to *@maze, game to *@game, java.io to *@io;
}
```

The above atom enhances the class `Door` with the new method `canOpen` using the open class pattern. This enhancement is realized by importing the previous implementation of the maze classes (`inmaze`), exporting an enhanced implementation (`outmaze`), and importing a fixed implementation of the maze classes (`maze`). The exported package `outmaze` inherits from the imported package `inmaze`, while the imported package `maze` inherits from the exported package `outmaze`.

Package inheritance is specified using the **extends** in much the same way that class inheritance is specified. When a package extends another package, the extending package does not need to be described with a package signature. The result of package inheritance is a blanket class extension of classes in the extending package with classes in the extended package. For example the atom *`edu.utah.maze.securable`*, the class `maze.Door` extends the class `outmaze.Door`, which extends the class `inmaze.Door`.

The open class pattern allows a package of classes to be extended by a unit by deferring the fixed implementation of those classes. Only the fixed implementation of the classes should be instantiated and referred to in unit signatures. In the atom *`edu.utah.maze.securable`*, only maze package to be bound to package parameters is the imported package `maze`. Classes in packages that are extended are always abstract to the Java code of an atom even if they do not have the abstract class modifier in their signatures. This ensures that classes from the packages `inmaze` or `outmaze` are not instantiated. Since the imported package `maze` is not extended by another package, classes in that package can be instantiated if not marked as abstract in their signatures.

The open class pattern can be used add new methods to a class, as well as override its existing methods. Consider the Java source code for the class `outmaze.Door` located in the Java source file `tutorial/edu.utah.maze.securable/outmaze/Door.java`:

```
package outmaze;
public abstract class Door extends inmaze.Door {
  public boolean canOpen(game.Player p) {
```

```
    return true;
  }
  public boolean canPass(game.Actor p) {
    if (!super.canPass(p)) return false;
    else if (p instanceof game.Player &&
     !this.canOpen((game.Player) p)) return false;
    else return true;
  }
  public Door() {
    super();
  }
}
```

The above source code enforces that a player cannot pass through a door without being able to open it. While the `canOpen` method initially returns true, as we will see this method will be overridden by classes in other units.

As we will see later in Section 4.4.1, package inheritance can be propagated through compounds.

4.4 Variables

Variables are used to make units in Jiazzi more reusable by making units more parametric. Whereas packages encourage large-scale reuse through coarse-grained linking, variables are more fine grained and can be used to "flush-out" a unit.

The first kind of variables we will discuss are lexemes (Section 4.4.1). Lexemes parameterize the names of classes and class members in the imported and exported packages of a unit. The second kind of variables we will discuss are generics (Section 4.4.2). Generics are sort of like classes in imported packages, except they are connected at a fine-granularity and cannot be instantiated or inherited from within a unit. This makes connecting generics "easier" since less type checking rules apply to them.

Neither generics nor lexemes require changes to the Java language used to construct the Java source code of an atom. Rather they can only be manipulated at the unit definition level. However, some conventions are used in the Java source code of an atom that we will discuss in the subsections.

4.4.1 Lexemes

Lexemes can be declared in both units and package signatures. The lexemes of a package signature are bound in bind lexeme clausess that behave in a way that is similar to bind package clauses. The lexemes of a unit are linked by compounds when the unit is instantiated into a local unit in link lexeme clauses that are similar in structure to link package clauses.

Consider the following package signature *`maze.secure.sig`*:

```
signature maze.secure = z : maze.base + {
```

```
  lexeme Item, Kind;
  package maze, game, io;
  bind package maze to *@maze, game to *@game, io to *@io;
  class [Kind]Door extends maze.Door {
    [Kind]Door();
    void set[Item](maze.[Item] item);
    protected maze.[Item] needed[Item]();
  }
  class [Item] extends game.Item {
    [Item]();
  }
}
```

The above package signature is parameterized by two lexemes, `Item` and `Kind`. A lexeme can be embedded in any identifier that is used to name a class, a non-constructor class member. A lexeme is embedded into an identifier using a pair of brackets ([]). For example, the methods `set[Item]` and `needed[Item]` described in the class `[Kind]Door` are both parameterized by the lexeme `Item`.

Lexemes declared in package signature are bound in unit signatures and composing package signatures through bind lexeme clauses that are very similar in format to bind package clauses. Unit signature also can declare lexemes, Consider the atom *edu.utah.maze.secure*:

```
atom edu.utah.maze.secure {
  lexeme Item, Kind;
  import inmaze : maze.securable;
  export outmaze extends inmaze : maze.secure;
  import maze extends outmaze;
  import game : game.securable;
  bind lexeme
    [Item] to outmaze@Item, [Kind] to outmaze@Kind;
  bind package
    maze to *@maze, game to *@game,
    java.io to *@io;
}
```

The above atom uses the open class pattern to add the classes `[Kind]Door`and `Item` to the a package of maze classes. The atom defines two lexemes, `Item` and `Kind`. These lexemes are bound to the lexemes with the same name in the exported package `outmaze`.

The source of a bind lexeme clause statement is an identifier that can have an embedded lexeme of the enclosing package signature or unit signature. The destination of a bind lexeme clause statement is a lexeme of a composed package signature or imported/exported package. As with bind package clauses, wild cards can be used in bind lexeme clauses.

To the Java source code of an atom, the brackets that are used to denote an embedded lexeme are

absent, leaving just the name of the lexeme embedded in the identifier. Consider the Java soruce code to the class `outmaze.[Kind]Door` exported from the atom *`edu.utah.maze.secure`*:

```
public abstract class KindDoor extends maze.Door {
  maze.Item item;
  public void setItem(maze.Item item) {
    this.item = item;
  }
  protected maze.Item neededItem() {
    return this.item;
  }
  public boolean canOpen(game.Player p) {
    if (!super.canOpen(p)) return false;
    if (this.item != null) {
      if (!p.containsItem(this.item)) {
return false;
      }
    }
    return true;
  }
  public KindDoor() { super(); }
}
```

In the above Java source code, the class `[Kind]Door`is declared simply as `KindDoor` in the Java source file `tutorial/edu.utah.maze.secure/outmaze/KindDoor.java`. Also, the method `set[Item]` is defined as the method `setItem`. To prevent ambiguity at the Java source code level, name clashes are always checked without considering the brackets of an identifier with an embedded lexeme.

Lexemes can be bound to identifiers that do not further embed lexemes. Consider the package signature *`maze.magic`*:

```
signature maze.magic = z : maze.secure + {
  package maze, game, io;
  bind package maze to *@maze, game to *@game, io to *@io;
  bind lexeme Magic to z@Kind, Spell to z@Item;
}
```

In the above package signature, the lexemes `Kind` and `Item` of the composed package signature *`maze.secure`* (identified by `z`) are respectively bound from the identifiers “Magic” and “Spell.” No class or class member names described by the package signature *`maze.magic`* have embedded lexemes. For example the classes `MagicDoor` and `Spell` are described by the package signature, not the classes `[Kind]Door` and `[Item]`. Lexemes can also be bound to an empty string using two double quotes in the source position of a bind lexeme clause statement.

The lexemes of a unit must be linked to when the unit is instantiated to create a local unit. The linking of lexemes are specified in link lexeme clauses that are similiar in format to link

package clauses. Consider the following package signature *maze.magic* and compound *edu.utah-.maze.magic*:

```
compound edu.utah.maze.magic {
  import inmaze : maze.securable;
  export outmaze extends inmaze : maze.magic;
  import maze extends outmaze;
  import game : game.securable;
  bind package
    maze to *@maze, game to *@game,
    java.io to *@io;
} {
  local
    s : edu.utah.maze.secure;
  link lexeme
    Magic to s@Kind, Spell to s@Item;
  link package
    inmaze to s@inmaze, s@outmaze to outmaze,
    maze to s@maze, game to s@game;
}
```

The above compound instantiates the atom *edu.utah.maze.secure* into the local unit `s` and links the identifiers "Magic" and "Spell" to its lexemes. The result is a compound that exports the classes `outmaze.MagicDoor` and `outmaze.Spell`.

The compound *edu.utah.maze.magic* also demonstrates how the open class pattern can be propagated through a compound. The package `inmaze` imported into the compound is linked to the package `inmaze` imported into the local unit, the package `outmaze` exported from the local unit is linked to the package `outmaze` exported from the compound, and the package `maze` imported into the compound is linked to the package `maze` imported into the local unit. Additionally, the compound adheres to the open class pattern by having the package `maze` extend the package `outmaze`, and the package `outmaze` extend the package `inmaze`.

4.4.2 Generics

Our last language feature is the generic. A generic is like an imported class, except that it cannot be instantiated or subclassed within a unit, and it is not connected at a package granularity. As a result, generics can be connected to a wider variety of classes and are suitable to be used in the representation of generic collection classes. Consider the following package signatures *generic* and *list*:

```
signature generic = {
  lexeme Generic;
  package generic;
  class [Generic] extends Object { (); }
```

```
}
signature list = g:generic + {
  lexeme List, Prefix;
  generic TYPE;
  package list, util;
  bind package list to g@generic;
  bind lexeme [List] to g@Generic;
  class [List] {
    [List](); int size[Prefix]();
    void add[Prefix](int index, TYPE item);
    boolean contains[Prefix](TYPE item);
    util.Iterator iterator[Prefix](); ...
  }
}
```

The package signature *generic* describes an empty class whose name is completely unspecified using the lexeme `Generic`. The package signature *list* composes *generic* and binds its lexeme `List` the the lexeme `Generic` of *generic*. An additional lexeme `Kind` is used in *list* to prefix methods added by the class that is completely parameterized by the lexeme `List`.

The package signature *list* declares a generic `TYPE`, which it uses as a placeholder for types that are stored in and retrieved from the `[List]` class. Generics can only be used as types in method and field declarations, they cannot be used as superclasses in the package signature.

Generics are bound in bind generic clauses that are similar in format to bind package clauses and bind lexeme clauses. Generics can also be declared in unit signatures. Consider the following atom *edu.utah.list*:

```
atom edu.utah.list {
  generic TYPE : java.lang.Object;
  lexeme List, Prefix;
  import inlist : generic;
  export outlist extends inlist : list;
  import list extends outlist;

  bind generic TYPE to outlist@TYPE;
  bind package list to inlist@generic,
    list to outlist@list, java.util to *@util;
  bind lexeme [List] to outlist@List,
    [List] to inlist@Generic,
    [Prefix] to outlist@Prefix;
}
```

The above atom uses the open class pattern to add list methods to a single class. It declares the generic `TYPE` and binds this generic to the generic `TYPE` in the package `outlist`. Generics have

bounds that are classes visible in the unit. In the above atom, the generic `TYPE` is bounded by the class `Object`, which is also the default if a bound is not explicitly specified. When classes are referred to in a unit signature or the linking section of a compound, a dollar symbol ($) must be used to separate the inner from their outer classes, e.g., if the class `Inner` is an inner class of `Outer`, and the class Outer is in the package `pkg`, the class can be referred to as `pkg.Outer$Inner`. Classes must be referred to as fully qualified, with the only exception being classes that originate from the package `java.lang`, e.g., the class `java.lang.Object` can be referred to simply as `Object`.

The Java source code of an atom can refer to generics by their names in the package `var`, where a stub class is generated for each generic of the unit. Consider the partial Java source code for the class `outlist.[List]` exported from the atom ***edu.utah.list***:

```
package outlist;

public abstract class List extends inlist.List {
  protected var.TYPE items[] = new var.TYPE[1];
  public void addPrefix(int index, var.TYPE item) {
    this.ensureCapacity(this.size + 1);
    for (int i = index; i < this.size; i++)
      this.items[this.size - i + index] = this.items[this.size - i + index - 1];
    this.items[index] = item;
    this.size++;
  }
  ...
}
```

The above Java source code can create an array of the generic `TYPE` by instantiating an array of the stub class `var.TYPE`. The stubs created to represent generics have private constructors (so they cannot be instantiated) and are final (so they cannot be subclassed). Because of this, generics can be linked to classes with a far greater latitude than classes that are imported in packages. As long as the generic's bound is satisfied, any class can be linked to a generic, whereas imported classes have restrictions because subclassing and instantiation must be accounted for. For example, a final class cannot be linked to an imported non-final class, and a class cannot be linked to an import of an interface, but no such restrictions exists for generics.

The result of the "mixin" list provided by the atom ***edu.utah.list*** is that we can add a list to a class several times, with different prefixes (using lexemes) and different types (using generics). Consider the following package signature ***maze.lists***:

```
signature maze.lists = z:maze.base + p:list + i:list + {
  package maze, game, io, util;
  bind package maze to z@maze, game to z@game, io to z@io;
  bind package maze to *@list, util to *@util;

  bind lexeme Room to *@List, Actor to p@Prefix, Item to i@Prefix;
  bind generic game.Actor to  p@TYPE, game.Item to i@TYPE;
```

```
}
```

The package signature *maze.lists* composes the package signature *maze.base* from Section 4.1.1 and the package signature *list* twice to add list methods to the class `Room` twice under two different prefixes: Actor and Item. As a result, a room is embedded with two lists: the first list keeps track of actors that are in the room while the second list keeps track of items in the room.

The generic `TYPE` of the package signature *list* are bound to different types depending on composed package signature. The composed package signature identified by `p` has `TYPE` bound from the class `game.Actor`. The composed package signature identified by `i` has `TYPE` bound from the class `game.Item`.

The result of the package signature *maze.lists* is the class `Room` that has two lists embedded in it. One list is accessed using the prefix "Actor" and contains objects of type `game.Actor`. The other list is accessed using the prefix "Item" and contains objects of type `game.Item`. Consider the compound *edu.utah.maze.lists*:

```
compound edu.utah.maze.lists {
  import inmaze : maze.base;
  export outmaze extends inmaze : maze.lists;
  import maze extends outmaze;
  import game;
  bind package maze to *@maze, game to *@game,
    java.io to *@io, java.util to *@util;
} {
  local
    ra : edu.utah.list, ri : edu.utah.list;
  link package
    inmaze to ra@inlist,
    ra@outlist to ri@inlist,
    ri@outlist to outmaze;
  link package
    maze to *@list;
  link lexeme Room to *@List,
    Actor to ra@Prefix, Item to ri@Prefix;
  link generic
    game.Actor to ra@TYPE,
    game.Item to ri@TYPE;
}
```

The above compound meets the requirements for the package signature *maze.lists*. It also demonstrates link generic clauses, which are similiar in format to link package clauses and link lexeme clauses.

This concludes are tutorial. The advanced maze game is defined in the compound *edu.utah.maze.game.two*, but before it can be linked, the Java source code for all maze games atoms must

compiled so the atoms can be linked. After the compound *edu.utah.maze.game.two* is linked, the maze game can be run as an application by executing the class `main.Main` in the JAR file `tutorial/edu.utah.maze.game.two.jar`.

Chapter 5

Technical

5.1 Identifiers

A *simple identifier* (sid) begins with an underscore or letter (e.g., `_foo8`), and continues with letters, numbers, and underscores. A *complex identifier* (cid) is a series of simple identifier separated by dots (e.g., `edu.utah._foo8`). An *embedded identifier* (eid) is like a simple identifier except a lexeme (which is itself a simple identifier) can be embedded in it using brackets (e.g., `set[Prefix]`). An embedded identifier can also be just a lexeme surrounded by brackets (e.g., `[List]`).

5.2 Syntax

The following sections describe Jiazzi's syntax. The following conventions are used:

- A non-terminal is in roman font, e.g., the non-terminal package-signature.
- An ascii token is in bold text type font, e.g., the ascii token **signature**.
- A non-ascii token is inside <>, e.g., a comma as <,>.
- A terminal identifier is in superscript font, and is qualified with :: as either a sid (simple identifier), cid (complex identifier), or eid (embedded identifier), e.g., the terminal SIGNATURE-NAME::cid is a complex identifier.
- When comma tokens are used inside star expressions, they are only active in each repetition of the star after the first, e.g., (<,> type)* can expand to type <,> type.

5.2.1 Signatures

A *package signature* describes the structure of classes in a package, and has the following syntax:

package-signature **::= signature** SIGNATURE-NAME::cid **<=>**
(COMPOSED-ID::sid **<:> SIGNATURE-NAME::**cid **<+>)* <{>**

```
  (((lexeme (<,> LEXEME::sid)*)
    (generic (<,> GENERIC::sid)*)
    (package (<,> PACKAGE-PARAMETER::sid)*)
    bind-package-clause | bind-lexeme-clause |
    bind-generic-clause | class-signature) <;>)*
<}>
```

A *class signature* describes the structure of a class within the context of a package signature, and has the following syntax:

```
class-signature ::= flags (class | interface) CLASS-NAME::eid
    (extends (<,> class-type)*)? (implements (<,> (class-type))*)? <{>
    ((method-signature) | (field-signature) |
     (constructor-signature) | (class-signature))*
<}>

method-signature ::= flags (type | void) METHOD-NAME::eid arg-throws<;>

constructor-signature ::= flags (CLASS-NAME::eid)? arg-throws<;>

arg-throws ::= <(>(<,> type (ARG-NAME::sid)?)*<)> (throws (<,> (class-type))*)?<;>

field-signature ::= flags type FIELD-NAME::eid<;>

flags ::= (public | protected | abstract | final | static)*

type ::= primitive | class-type | type<[]> | TYPE-VARIABLE::sid

class-type ::= Object | String | PACKAGE-PARAMETER::sid <.> CLASS-NAME::eid
```

A *unit signature* dedefines a unit's public interface, including the structure of its imported and exported packages. It has the following syntax:

```
unit-signature ::= UNIT-NAME::cid <{>
  ((((import | export) (<,> (package-decl))*) |
    (generic (, generic-decl)*) |
    (lexeme (, LEXEME::sid)*) |
    bind-package-clause |
    bind-lexeme-clause |
    bind-generic-clause) <;>)*
<}>

package-decl ::= PACKAGE-NAME::cid (<:> SIGNATURE-NAME::cid)?
             (extends PACKAGE-NAME::cid)?

generic-decl ::= GENERIC::sid (<:> class-type)?
```

class-type ::= PACKAGE-NAME::cid <.> CLASS-NAME::eid

unit-def ::= (atom unit-signature) | (compound unit-signature linkage-section)

5.2.2 Units

A *linkage section* has the following syntax:

```
linkage-section ::= <{>
  (((local (, (LOCAL-NAME::sid : UNIT-NAME::cid))*) |
    link-package-clause |
    link-lexeme-clause |
    link-generic-clause) <;>)*
<}>
```

A *raw unit* contains an atom's Java source and class files that result from source compilation. A raw unit is a directory located in the project path directory with the same name as the atom it corresponds to. An atom's corresponding raw unit is used to create its corresponding linked unit.

The result of linking a unit is its corresponding *linked unit*. Physically a linked unit exists as a Java archive (JAR) file in the project path directory with the same name as the unit it corresponds to. A linked unit are recreated whenever the linker is executed on the unit it corresponds to and the linked unit is out of date.

5.2.3 Bind Clauses

Bind clauses bind the parameterized elements of package signatures. All bind clauses have the following syntax:

bind-clause ::= bind bind-type (<,> source to sink)*

bind-type ::= package | name | type

sink ::= (sink-target | <*>) <@> sink-param

sink-target ::= COMPOSE-ID::sid | PACKAGE-NAME::cid

The sink-target non-terminal is a COMPOSED-ID::sid if the bind-clause is used in a package signature, otherwise it is a PACKAGE-NAME::cid that identifies a package that is imported or exported in the enclosing unit signature. When a wild card (*) is used instead of a sink in a bind clause, the bind clause statement matches all composed package signatures, if the bind clause is in a package

signature, or all imported and exported packages, if the bind clause is in a unit package signature, that have the sink-param specified in the bind clause.

A *bind package clause* binds the package parameters of a package signature. It has the following syntax as a refinement of bind-clause:

source-param ::= PACKAGE-PARAMETER::sid | PACKAGE-NAME::cid

sink-param ::= PACKAGE-PARAMETER::sid

The non-terminal source is a PACKAGE-PARAMETER::sid when the bind package clause is used in a package signature, where the identified package parameter is declared in the enclosing package signature. Otherwise source is a PACKAGE-NAME::cid, where the identified package is either imported or exported into the unit, or is accessible through the CLASSPATH. The non-terminal sink-param identifies a package parameter declared by the package signature used in the definition of source-target.

A *bind lexeme clause* binds the lexemes of a package signature. It has the following syntax as a refinement of bind-clause:

source-param ::= EXPANDED::eid | <‘‘’’>

sink-param ::= NAME-VARIABLE::sid

The non-terminal source-param is either an embedded identifier that can embed a lexeme declared by the enclosing unit signature or package signature, or it is the empty string. The non-terminal sink-param identifies a lexeme that is declared by the package signature used in the definition of source-target.

A *bind generic clause* binds the generics of a package signature. It has the following syntax as a refinement of bind-clause:

source-param ::= class-type | GENERIC::sid

sink-param ::= GENERIC::sid

The non-terminal source-param is either a class-type, or a generic that is declared in the enclosing package signature or unit signature. Note that the non-terminal class-type has different definitions in unit signatures and package signatures. The non-terminal sink-param identifies a lexeme that is declared by the package signature used in the definition of source-target.

5.2.4 Link Clauses

Link clauses are very similar to bind classes except that they are only located in a compound's linkage section and link parameterized elemenets of local units. All link clauses have the following syntax:

link-clause ::= link link-type (<,> source to sink)*

link-type ::= package | name | type

sink ::= (sink-target <@>)? sink-param

sink-target ::= LOCAL-NAME::sid | <*>

The sink-target can identify a local unit or use a wild card, in which case it matches all local units that ahve the specified sink-param. In the non-terminal sink, sink-target is only optional in package link clauses.

A *link package clause* links the exported packages of local units and imported packages of the enclosing compound to the imported packages of local units and exported packages of the enclosing compound. It has the following syntax as a refinement of link-clause:

source ::= (source-target <@>)? source-param

source-target ::= LOCAL-NAME::sid

source-param ::= PACKAGE-NAME::cid

sink-param ::= PACKAGE-NAME::cid

If ta source-target is not specified, then the source-param identifies a package imported into the enclosing compound, otherwise source-param identifies a package exported from the local unit identified by source-target. If a sink-target is not specified, then the sink-param identifies a package exported into the enclosing compound, otherwise sink-param identifies a package imported from the local unit identified by sink-target. If sink-target is a wild card, then it identifies imported packages of the compound's local units and a compound's exported package, that can be identified with sink-target.

It is not valid for both the sink-target and source-target to be unspecified. This would entail a connection from a compound's import to a compound's export, which doesn't make sense.

A *link lexeme clause* links the lexemes of a local unit. A sink-target is mandatory as the compound's own lexemes cannot be linked inside the compound. It has the following syntax as a refinement of link-clause:

source ::= EXPANDED::eid | <‘‘’’>

sink-param ::= LEXEME::sid

The source is either an embedded identifier that can embed a lexeme declared by the enclosing compound, or the empty string. The sink-param is a lexeme declared by the unit used to create the local unit identified by sink-target.

A *link generic clause* links a class type to the type variables of a local unit.

source `::= GENERIC::`sid `|` `((LOCAL-NAME::`sid `<@>)?` class-type)

sink-param `::= GENERIC::`sid

The source is either a generic declared by the enclosing compound, or a reference to a class, in which case it can be a reference to a class that is imported into the compound, exported from a local unit, or accessible through the CLASSPATH. The sink-param is a generic declared by the unit used to create the local unit identified by sink-target.

5.3 Semantics

package signature to unit translation

unit to local unit translation

5.4 Type Checking

unit type checking

atom type checking

compound type checking

package matching

class matching

type variable matching

name clashes

5.5 Tools

Tools and the tutorial are accessible from the *Jiazzi distribution directory*, which is the directory that is created when the Jiazzi distribution is unarchived. See Chapter 3 for details on how to install Jiazzi.

There are only two tools in Jiazzi. The first tool generates stubs Java classes so Jiazzi can be used with conventional Java compilers. Both tools are executed through the Java program `jiazzi.Main` and take the form:

```
java jiazzi.Main [project path] -<tool-name> <unit-name>
```

`tool-name` is the name of the tool to be executed, which right now can be either `stub` for stub generator or `link` for linker. Both tools require a `unit-name`, which becomes the subject of the tool, and will fail with an exception if the unit definition is malformed.

The *project path* is where all unit definition files, package signature files, raw units, and linked units are located. Right now, only one project path can be specified.

The *stub generator* operates on units that are atoms. Atoms are units that do not link other units, and instead are built directly from conventional Java code. Units can import classes that are provided by other units. Because standard Java compilers do not know about these imported classes, the stub generators creates empty classes using the known class signatures of classes in imported packages so a Java compiler can be used compile the Java source code of the atom.

When executed, a stub generator creates a directory with the name of the unit in the project path if it does not exist already, which contains the atom's raw unit. In the raw unit, additional directories are created for exported packages if they do not exist already. Java source files are then generated according to the class signatures of classes in exported packages. Java source files generated for exported classes are known as skeletons, and are never overwritten by the stub generator. Java class files for classes of imported packages as well as generic classes are generated using class signatures and placed in the file `stubs.jar` within the unit's directory. The `stubs.jar` file of an atom is regenerated each time the stub generator is run.

, and must be included in the classpath when an atom's raw unit is compiled using a conventional Java source file.

The Java source code for a raw unit can be compiled using a conventional *Java source compiler*, such as `javac.exe`. Source compilation occurs by changing to the directory of the atom's corresponding raw unit (`project-path/unit-name`), and running the compiler on the Java source files in that directory. The `stubs.jar` file of the atom must be added to the classpath when the source compiler is run, e.g., `javac.exe -cp ./stubs.jar`. As the atom is separated from other units in the system, it is practical to run the Java source compiler on all Java source files of the raw unit at once. Assuming the currect directory is the atom's raw unit directory, just execute `javac.exe -cp ./stubs.jar */*.java`.

The *linker* translates units into linked units, performing necessary type checking (Section 5.4) along the way. If the unit is an atom, then the linker processes classes from the atom's corresponding raw unit. If the unit is a compound, then all units linked in the compound are first linked, duplicated to create local units, and rewrite according to the compound's linkage section.

The linker will only link a unit if its corresponding linked unit does not exist or is out of date with respect to the sources and binaries used in the linking. When linking a compound, units linked in the compound are recursively linked if they are out of date. The benefit of this is that the linker need only be run on a top-level compound and does not need to be run explicitly on individual units linked in the compound.

Chapter 6

Troubleshooting

This section is blank right now. We can add to it as people use Jiazzi and run into problems.

Bibliography

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

[5] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proc. of OOPSLA*, pages xxx–yyy, Oct. 2001.