

Aspect-Oriented Programming with Jiazzi

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

ABSTRACT

We present aspect-oriented programming in Jiazzi. Jiazzi enhances Java with separately compiled, externally-linked code modules called *units*. Besides making programming in Java generally more modular, units are also effective “aspect” constructs that can separate concerns. The unit-linking metaphor provides a convenient and explicit way for programmers to explicitly control the inclusion and configuration of code that implements a concern, while separate compilation of units enhances concern independent development and deployment. The expressiveness of concern separation are enhanced by units in two ways. First, classes can be made open to the addition of new fields and methods by multiple units, which enables the direct modularization of concerns that crosscut objects. Second, the signatures of methods and classes used in a unit can be made open to refinement by other units, which makes it easier to integrate concern implementations by isolating them from the naming and calling requirements of shared methods and classes.

1. Introduction

Jiazzi [17] is an enhancement of Java that adds support for encapsulated code modules known as *program units* [8]. Units were originally designed to make programming more modular by providing for the explicit and safe management of code modules. This heritage also makes units ideal constructs to support *aspect-oriented programming* [13] (AOP), which focuses on modularizing programming concerns not easily modularized by classes or other traditional modularity constructs.

Units in Jiazzi contain the code for multiple Java class implementations, which is an ideal granularity for modularizing concerns whose implementations involve multiple classes. Units are linked together through the use of an expressive linking language. This linking language acts as Jiazzi’s aspect con-

figuration language, where the inclusion and configuration of code that implements a concern amounts to unit linking. Units undergo **separate compilation** [2]: the internal implementations of units are compiled and type-checked independent of how they will be linked. Separate compilation promotes the separate reasoning, independent development, and binary deployment of code that implements concerns.

Units in Jiazzi directly facilitate concern modularization in two ways. First, units enable the creation of *open classes* [5], which are classes that can be enhanced with new methods and fields without editing their source code or breaking their existing subclassing relationships. Units can create new open classes or enhance existing open classes with new implementation, which allows units to modularize concerns whose implementations cross-cut object and class boundaries. Second, units support *open signatures*, where details necessary for the use of methods and classes visible in a unit can be refined as the unit undergoes linking. In object-oriented languages such as Java, these details are class and method names, and method argument lists. By keeping its signature open, a concern code can be composed with a wider range of other concern code whose signatures may vary with respect to shared classes and methods. Neither open classes nor open signatures interfere with unit separate compilation.

AOP in Jiazzi can easily be adopted into existing Java program development practices. Jiazzi adds a separate linking language to Java and does not require an extension to the syntax of the core Java language. Instead, units are constructed out of standard Java source code, that can be compiled using standard Java source compilers. Unit linking is implemented by rewriting Java bytecode, so the classes provided by linked unit implementations can execute on conventional Java virtual machines.

This paper concentrates on how Jiazzi can be used in AOP, rather than the details behind the design of Jiazzi’s unit model. The rest of this paper is organized as follows. Section 2 introduces Jiazzi’s unit model and linking language. Section 3 describes how open classes are used in Jiazzi to modularize object crosscutting concerns. Section 4 describes how open signature are used in Jiazzi to make concern implementations more generic and reusable. Section 5 discusses type checking and implementation in Jiazzi. Section 6 presents related work and Section 7 summarizes our conclusions.

```
signature mzbase = {
  class Maze extends Object { Maze(); ... }
  abstract class Entity extends Object
  { Entity(); abstract void display(); ... }
  class Room extends Entity
  { Room(); Item item(int n); ... }
  class Door extends Entity
  { Door(); boolean enter(Player p); ... }
  class Player extends Entity
  { Player(String name); void exec(); ... }
  class Item extends Entity { Item(); ... }
}
```

Figure 1: The package signature `mzbase` describes a package of basic maze-game classes.

```
signature program = {
  class Main extends Object
  { static void main(String args[]); }
}
atom driver
{ import maze : mzbase;
  export main : program; }
// file: driver/main/Main.java
package main;
class MyMaze extends maze.Maze {...}
public class Main extends Object {
  public static void main(String args[])
  { maze.Maze maze = new MyMaze();
    maze.Player p = new maze.Player(args[0]);
    maze.Room rooms[] = {...};
    maze.Door doors[] = {...};
    ... p.exec(); ... } }
```

Figure 2: The package signature `program`, the atom driver and Java source code of the atom driver.

2. Jiazzi Overview

This section describes much of what we have already published about Jiazzi [17]. Since this paper focuses on the usability of Jiazzi for AOP, the syntax presented in this paper has more features than previous work. For a more in-depth discussion of Jiazzi’s unit model, including the details behind its mechanisms and implementation, see our OOPSLA 2001 paper [17]. We describe Jiazzi by using it to construct a **maze game** [9, 10] software application. The basic version of this maze game involves a player exploring a maze of rooms, which are connected together by doors and populated with items.

A basic maze game can be divided into a *package* in Jiazzi with the following core classes: `Maze`, `Entity`, `Room`, `Door`, `Player`, and `Item`. A package in Jiazzi is similar to a package in Java: both are constructs that group classes together. The basic structure of these maze-game classes are described by the *package signature* `mzbase` in Figure 1. Package signatures describe the classes in a package independently of their implementations.

Modules of Java code in Jiazzi are encapsulated into *units*. In Figure 2, a maze-game application driver is encapsulated in the *atom driver*. An atom is a kind of unit that is constructed directly from Java source code. The *atom driver imports* the package `maze`, which creates a dependency on the basic maze-game classes that must be provided by another unit. The specific implementations of the basic maze-game classes imported

```
atom base
{ export maze : mzbase; }

compound game
{ export main : program;
  export maze : mzbase;
  link unit base, driver; }
```

Figure 3: The atom base and the compound game; the Java source code implementation of base is not shown.

in `maze` are not known when the Java source code of `atom driver` undergoes development: the structure of these classes is described by the package signature `mzbase` from Figure 1. The atom driver *exports* the package `main`, which provides an application entry-point class to other units. The implementation of `atom driver` is hidden from its clients: the structure of the class `Main` exported in package `main` is described to clients by the package signature `program`.

The Java source code of `atom driver` can refer to basic maze-game classes imported in the package `maze` as if they were normal Java classes. In the Java source code implementation of the `atom driver` shown at the bottom of Figure 2, the imported basic maze-game classes are used as types, instantiated using the `new` operator, and extended using inheritance. Conversely, the implementation of the class `Main` in exported package `main` must conform to its description in the package signature `program`.

Linking in Jiazzi specifies which unit will provide the implementation of the basic maze-game classes to the `atom driver`. This linking occurs in the unit `game`, which is a *compound*. A compound is a kind of unit that is constructed by linking other units together. The provider of the basic maze-game classes to the `atom driver` is the `atom base`, which exports the maze-game classes in its package `maze` described by the package signature `mzbase` from Figure 1. Linking occurs by specifying the atoms `base` and `driver` in the `link unit` clause of the compound `game`.

Connections between imported and exported packages of the units linked in the compound `game` are established automatically using package name matching.¹ The package `maze` exported from the `atom base` is connected to the package `maze` imported into the `atom driver`. The result of this connection is that any uses of the basic maze-game classes in the Java code of the `atom driver` become uses of the basic maze-game classes implemented in the `atom base`.

All the connections between packages established in compound `game` are illustrated in Figure 4. Besides making connections between linked units, the packages `maze` and `main`, exported from the atoms `base` and `driver`, respectively, are both connected to packages exported from the compound `game`. As a result, these packages can be provided to units when the compound `game` itself is linked by other compounds. Because the compound `game` does not import any packages, the Java classes it contains can be loaded directly into a Java virtual machine. By providing the executable class `Main` in the package `main`,

¹Connections can always be specified manually using linking syntax not used in this paper. See the Jiazzi manual [16] for details.

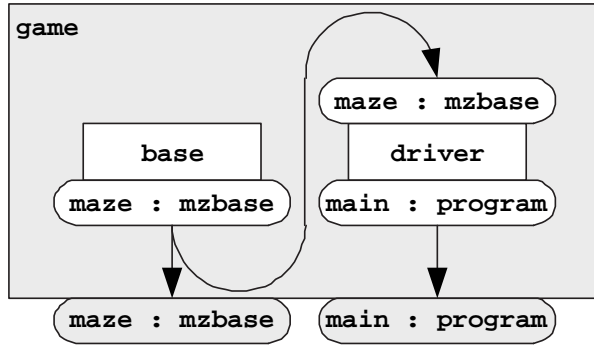


Figure 4: An illustration of the linking that occurs in compound game; imported packages are on top of unit rectangles; exported packages are on bottom of unit rectangles; arrows point from connection sources to sinks.

```
signature mzmagic = mzbases + {
  class Spell extends Item { Spell(); }
  class Door +=
  { Spell neededSpell();
    void setSpell(Spell spell); ... }
  class Player +=
  { void addSpell(Spell spell);
    boolean hasSpell(Spell spell);
    void castSpell(Spell spell); ... }
}
```

Figure 5: The package signature `mzmagic`, which is built out of package signature `mzbases`.

the compound `game` acts as a self-contained Java application.

3. Open Classes

Suppose the maze game is enhanced with a new *magic* “feature.” The *magic* feature requires players to find and cast spells to open some doors. The additional classes and methods that are added to the basic maze-game package to support the *magic* feature are described by the new package signature `mzmagic` in Figure 5, which uses the add (+) operator to add to the package signature `mzbases` from Figure 1. The package signature `mzmagic` describes structure already described by `mzbases`, adds a new description for the class `Spell`, and uses the accumulate (+=) operator to add new method descriptions to classes that are already described by the package signature `mzbases`.

For the sake of good modularity, the Java source code for the *magic* feature must be separated from the basic maze-game Java source code. However, this code must also add new implementation to the classes `Door` and `Player`, which already have basic maze-game implementations. Conventional inheritance cannot be used to enhance these classes because it suffers from an *extensibility problem* [7]: implementation added to a class by creating a new subclass is not also added to the original class’s existing subclasses. In *Jiazzi*, we solve these problems with *open classes* [5, 19], which are classes that can be enhanced with new implementation without modifying their original source code. New implementation added to an open class are also added to the open class’s existing subclasses. Open classes are implemented through linking in *Jiazzi*, and do not require changes to the Java language.

```
atom opmagic
{ open maze : mzbases -> mzmagic; }

// file: opmagic/maze/Door.java
package maze;
public class Door extends _super_Door {
  private Spell spell = null;
  public Door() { super(); }
  public void setSpell(Spell s) { spell = s; }
  public Spell neededSpell() { return spell; }
  public boolean enter(Player p)
  { if (neededSpell() != null)
    if (p.casting != this.neededSpell())
      return false;
    return super.enter(p); } ...
}

// file: opmagic/maze/Player.java
package maze;
public class Player extends _super_Player
{ Spell casting; ... }
```

Figure 6: The atom `opmagic` and Java source code of `opmagic` for the enhancement to open classes `Door` and `Player`.

The atom `opmagic` in Figure 6 adds new implementation to the *open package* `maze`, which by being declared with the `open` keyword, is a package of open maze-game classes. Unlike a package that is imported or exported, the open package `maze` is described by two package signatures separated by an arrow (->); the first package signature `mzbases` from Figure 1 describes the base maze-game classes as its *imported structure*, while the second package signature `mzmagic` from Figure 5 describes the *magic* feature enhanced base maze-game classes as its *exported structure*.

Shown in Figure 6, the Java source code for the open class `Door` in atom `opmagic` can freely refer to the imported structure of the open package `maze`. To expose the imported structure of open class `Door` to a Java source compiler, the Java source definition of open class `Door` extends the class `_super_Door`, which is a special name automatically generated by *Jiazzi*. This allows the Java source code of open class `Door` to call and override the method `enter`, whose previous implementation is provided by an unknown unit. The implementation of the atom `opmagic` must satisfy the exported structure of the open package `maze`, such as by defining the methods `setSpell` and `neededSpell` in class `Door`.

Besides the *magic* feature, the maze game can be enhanced with other features, such as a *locked* feature where players must find and use keys to open some doors. The package signature `mzlocked` and the atom `locked` in Figure 7 describe and implement the *locked* feature in a manner similar to how the package signature `mzmagic` and the atom `opmagic` describe and implement the *magic* feature. The package signatures `mzmagic` and `mzlocked` are combined into the package signature `mzmagloc`, which describes the structure of maze-game classes enhanced with the composite *magic locked* feature. By composing both of these package signatures, `mzmagloc` describes a package of maze-game classes with all structure of the *locked* and *magic* features. That is, the package described has both classes `Spell` and `Key` and the class `Door` has both the methods `neededSpell` and `neededKey`.

```

signature mzlocked = mzbase + {
  class Key extends Item { Key(); }
  class Door +=
  { key neededKey();
    void setKey(Key key); ... }
  class Player +=
  { void addKey(Key key); boolean hasKey(Key key);
    void useKey(Key key, Door door); ... }
}
atom oplocked {
  open locked : mzbase -> mzlocked;
}
signature mzmagloc = mzmagic + mzlocked;

```

```

compound opmagloc {
  open maze : mzbase -> mzmagloc;
  link unit opmagic, oplocked;
}

```

Figure 7: The package signatures `mzlocked` and `mzmagloc`, the atom `oplocked`, and the compound `opmagloc`; the Java source code of atom `oplocked` is not shown.

```

atom opbase {
  open maze : empty -> mzbase;
}
compound game2 {
  export main : program;
  export maze : mzmagloc;
  link unit driver;
  link unit opbase, opmagloc;
}

```

Figure 8: The atom `opbase` and the compound `game2`; the Java source code for atom `opbase` is not shown.

The composite *magic locked* feature is implemented by linking the atoms `opmagic` and `oplocked` together in the compound `opmagloc` defined at the bottom of Figure 7. Connections between the imported and exported structure of the `maze` open packages of atoms `opmagic` and `oplocked` are established automatically by matching their names and by the order in which the linked units are specified in the compound’s link clause. Connecting these `maze` open packages together conceptually merges them into a single open package that contains the maze-game implementations of both the *magic* and *locked* features. Because it determines the overriding order for methods that are overridden, the order that atoms `opmagic` and `oplocked` are linked is important. In compound `opmagloc` the atom `oplocked` is linked after the atom `opmagic`, so the implementation of the method `enter` of class `Door` is last overridden in the atom `oplocked`.

To form complete maze-game classes with both the *magic* and *locked* features, the compound `game2` in Figure 8 links the compound `opmagloc` with the atom `opbase`, which provides a basic implementation of the maze-game open classes. The imported structure of open package `maze` in `opbase` is described with the built-in package signature `empty`, which describes a package with no classes. The compound `game2` in Figure 8 links the units `opbase` and `opmagloc` together with the atom `driver` from Figure 2 to create a complete maze-game application. The compound `game2` does not have any imported or open packages, so its classes can be loaded directly into a Java virtual machine.

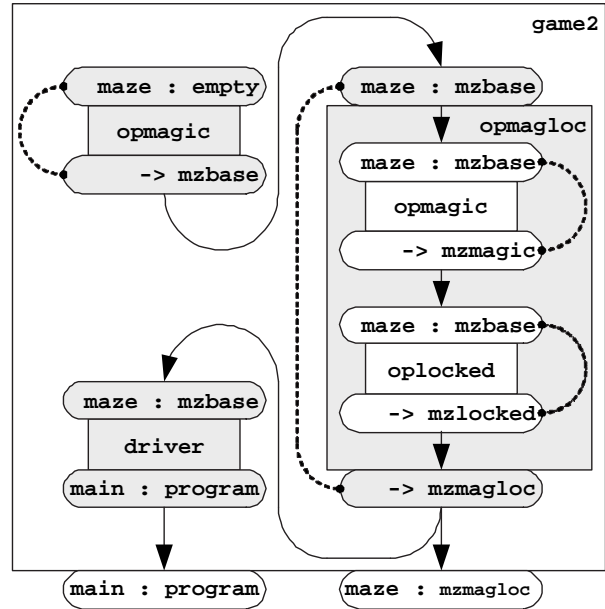


Figure 9: An illustration of the connections established in the linking of compounds `game2` and `opmagloc`; the top imported and bottom exported portions of an open package are connected together by a dashed handles to illustrate their interdependency.

Connections made in compounds `game2` and `opmagloc` are illustrated in Figure 9. Connections to the imported and exported open package `maze` structure of compound `opmagloc` in compound `game2` create indirect connections to and from the imported and exported open package `maze` structure of atoms `opmagic` and `oplocked`. The exported open package `maze` structure of compound `opmagloc` is connected to the imported `maze` package of atom `driver` and the exported `maze` package of compound `game2`. Connecting an open package to an imported or exported non-open package “closes” the classes in the open package: they can no longer be enhanced as open classes. Inside the atom `driver` and outside the compound `game2`, classes in the package `maze` do not appear as open, and no new implementation can be added from these contexts.

By selectively linking the atoms `opmagic` and `oplocked` within a maze-game application, other variations can be created that have only the *magic* feature, only the *locked* feature, or neither feature. Feature inclusion is controlled through linking specified by Jiazzi’s linking language, and source code is not modified when a new configuration of the maze game is created.

4. Open Signatures

While open classes can modularize code that cross-cuts class boundaries, separation of such code can be inhibited by the signatures of shared classes and methods that vary between units. In this section, we describe constructs that allow units to be reused when describing concerns.

```

signature mzsecure = mzbase + {
  name [DEVICE];
  class [DEVICE] extends Item { [DEVICE]() };
  class Door +=
  { [DEVICE] needed[DEVICE]();
    void set[DEVICE]([DEVICE] dvc); ... }
  class Player +=
  { void add[DEVICE]([DEVICE] dvc);
    boolean has[DEVICE]([DEVICE] dvc); ... }
}
atom opsecure {
  name [DEVICE];
  open maze : mzbase -> mzsecure;
}

```

Figure 10: The package signature `mzsecure` and the atom `opsecure`.

4.1 Name Parameters

There is significant overlap between the functionality of the maze game *magic* and *locked* features. Both features restrict access to doors to players who have used certain items. The implementations of the *magic* and *locked* features potentially overlap. However, the overlapping implementation is difficult to separate because names used between the features' implementations are different. For example, the item needed by a door is queried using the method `neededSpell` of class `Door` in the *magic* feature implementation, while the method `neededKey` is used in the *locked* feature implementation.

Forcing both feature implementations to agree on method and class naming would create irresolvable ambiguities; e.g., calling two distinct `needed` methods from the class `Door` is not possible in the Java language. Rather than force naming agreement, both *magic* and *locked* feature implementations can adhere to a common **naming convention**. The package signatures `mzmagic` (Figure 5) and `mzlocked` (Figure 7) both follow the same naming convention in naming methods added to the signatures of classes `Door` and `Player`, where a name is composed of a verb, which describes the action of the method, and a subject, which is the name of the item the method uses, e.g., “needed-Spell” and “needed-Key.”

This naming convention can be codified in Jiazzi using *open signatures*, which are unit signatures that are open to refinement. The package signature `mzsecure` in Figure 10 declares the *name parameter* `[DEVICE]` with the **name** keyword, where a name parameter can be used only as part of a class or method name: e.g., in the name of method `needed[DEVICE]` or class `[DEVICE]`. The name parameter `[DEVICE]` is a placeholder for the parts of method and class names that are unbound in the naming convention secure features. The atom `opsecure` in Figure 10 uses the package signature `mzsecure` to describe the open package `maze`. The `[DEVICE]` name parameter in atom `opsecure` that is used in package signature `mzsecure` is unbound, which allows the signature of atom `opsecure` to be refined later when it is linked with other units and `[DEVICE]` is bound.

Shown in Figure 11, the Java source code for atom `opsecure` can effectively provide the common implementation of the *magic* and *locked* features because the `[DEVICE]` name parameter isolates the source code of `opsecure` from the features' dif-

```

// file: opsecure/maze/Door.java
package maze;
public class Door extends _super.Door {
  DEVICE dvc;
  public void setDEVICE(DEVICE d)
  { dvc = d; }
  public DEVICE neededDEVICE() { return dvc; }
  ... }

// file: opsecure/maze/DEVICE.java
package maze;
public class DEVICE extends Item
{ public DEVICE() { ... } ... }

```

Figure 11: The Java source code for the open maze game classes `Door` and `[DEVICE]` in the implementation of atom `opsecure`.

```

signature mzmagic2 = mzsecure + {
  [DEVICE] = Spell;
  class Player +=
  { void castSpell(Spell spell); }
}
signature mzlocked2 = mzsecure + {
  [DEVICE] = Key;
  class Player +=
  { void useKey(Key key, Door door); }
}

```

Figure 12: The package signatures `mzmagic2` and `mzlocked2`.

ferent naming requirements. Method and class names parameterized by `[DEVICE]` appear as normal Java identifiers without the brackets. The method `needed[DEVICE]` can be referred to as `neededDevice`, while the class `[DEVICE]` can be referred to as `DEVICE`. Name parameters are only reasoned about and manipulated in Jiazzi's linking language. Any uses of a partial identifier `DEVICE` in the Java source code of atom `opsecure` do not require any special reasoning and do not create any strange interactions later when the name parameter `[DEVICE]` is bound.

The name parameter `[DEVICE]` in package signature `mzsecure` is given a value using the binding operator (`=`) when package signature `mzsecure` is used by the package signatures `mzmagic2` and `mzlocked2` in Figure 12. `[DEVICE]` becomes fixed identifier `Spell` in package signature `mzmagic2`, while it becomes the fixed identifier `Key` in package signature `mzlocked2`. These fixed identifiers replace uses of `[DEVICE]` in `mzsecure`. In `mzmagic2`, the class `[DEVICE]` is renamed as the class `Spell`, while the method `needed[DEVICE]` of class `Door` is renamed as the method `neededSpell`. The package signatures `mzmagic2` and `mzlocked2` describe classes and methods that are equivalent to those described in the package signatures `mzmagic` from Figure 5 and `mzlocked` from Figure 7, respectively.

The atom `oplocked2` in Figure 13 provides the maze game implementation of the *locked* feature that builds on an implementation of the secure feature. In the Java source implementation of `oplocked2`, the `[DEVICE]` name parameter is replaced with the fixed identifier `Key` identifier by binding in the package signature `mzlocked`, which describes the exported structure of open package `maze`. As a result, the class `Key` is imported in the open package `maze`, not the class `[DEVICE]`, and the method `neededKey` is imported in the open class `Door`, not

```

atom oplocked2 {
  open maze : mzsecure -> mzlocked2;
}
// file: oplocked2/maze/Door.java
package maze;
public class Door extends _super.Door {
  boolean isLocked;
  public boolean enter(Player p)
  { if (isLocked) return false;
    return super.enter(p); }
}
// file: oplocked2/maze/Player.java
package maze;
public class Player extends _super.Player {
  public void useKey(Key k, Door d)
  { if (this.hasKey(k) && d.neededKey() == k)
    d.isLocked = !d.isLocked; }
}

```

Figure 13: The atom `oplocked2` and its implementation of Java source code.

```

atom opmagic2
{ open maze : mzsecure -> mzmagic2; }

compound opmagloc2 {
  open maze : mzbase -> mzmagloc;
  link unit opsecure where [DEVICE] = Spell,
           opsecure where [DEVICE] = Key,
           opmagic2, oplocked2;
}

```

Figure 14: The atom `opmagic2` and the compound `opmagloc2`.

the method `needed[DEVICE]`.

The atom `opmagic2` in Figure 14 provides the implementation of the *magic* feature. The compound `opmagloc2` in Figure 14 links the atom `opsecure` twice to accommodate its reuse by both the atoms `opmagic2` and `oplocked2`. Each time `opsecure` is linked, its unbound `[DEVICE]` name parameter in `open package maze` is bound in a **where** clause: to the fixed identifier `Spell` to accommodate its linking with `opmagic2`, and to the identifier `Key` to accommodate its linking with `oplocked2`. Each linking of `opsecure` creates a duplicate of its implementation and replaces references to `[DEVICE]` in method and class names with its bound identifier. The linking and name parameter bindings that occur in compound `opmagloc2` are illustrated in Figure 15. Rewriting is performed over bytecode by Jiazzi's linker, and does not affect separate compilation because correct usage and implementation of methods and classes can be verified independently of their actual naming requirements. The compound `opmagloc2` is functionally equivalent to the compound `opmagloc` from Figure 7, and the former can be linked instead of the latter in the compound `game2` from Figure 8.

4.2 Argument Parameters

Displaying the maze game application is the responsibility of the method `display`, which is declared abstract by the class `Entity` and implemented by the various maze game classes. The method `display` is an example of a traversal method; when `display` called on an object, the object will call `display` on its sub-objects as appropriate; e.g., a call to `display` a room

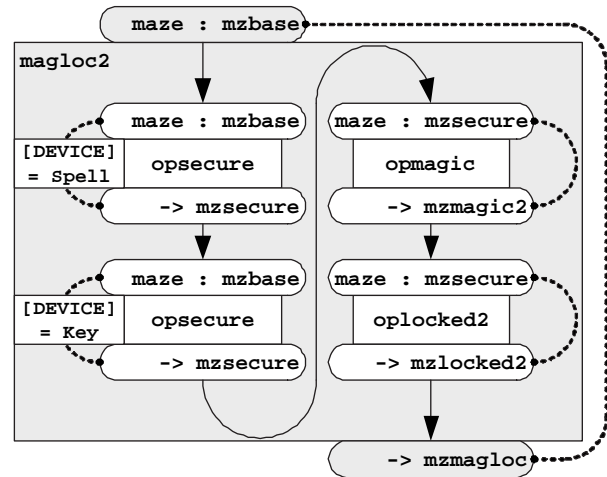


Figure 15: An illustration of the linking in compound `opmagloc2`; bindings of `[DEVICE]` name parameters is indicated next to the units they are bound for.

will cause calls to `display` on items in the room. Player actions also trigger `display` calls; e.g., when the method `enter` of class `Door` is called, calls to `display` are made to inform the player of the action's result.

Because it is not yet known in what way the maze game will be displayed when the basic maze game classes are implemented, the `display` methods can only be partially implemented in these classes. Code for the *cli* and *gui* features will determine if the maze game displays on a command line interface (*cli*) or graphical user interface (*gui*). This choice presents a dilemma: either choice for the display feature will require different arguments for the `display` method. The *cli* `display` method will require a stream interface to the command line, while *gui* will require a graphics context. As a result, arguments required by display feature code of the `display` method cannot be specified in the basic implementation of the maze game classes.

Solutions to this problem could abstract `display` method arguments as static and instance fields in various classes. These solutions must be carefully crafted because the maze game could be concurrent. For example, there could be multiple players using separate displays or `display` method arguments could change as they are passed to deeper `display` method calls. Ideally, new arguments could be added to the `display` method when the display feature to be implemented becomes known, but in Java this cannot be done without editing the source code of the original `display` declarations and definitions.

Besides codifying naming conventions, open signatures can be used in Jiazzi to add new arguments to methods after they have been declared and defined. The package signature `mzbase2` in Figure 16 declares the *argument parameter* `[DISPLAY]` with the **arg** keyword. Argument parameters can be used only after argument lists in method signatures. The argument parameter `[DISPLAY]` abstracts `display` method arguments. This includes the methods `exec` and `enter` in classes `Player` and `Door`, whose implementations can potentially call `display` methods. As a result, these two methods must have access to the

```
signature mzbbase2 = {
  arg [DISPLAY];
  class Maze extends Object { Maze(); ... }
  abstract class Entity extends Object
  { Entity();
    abstract void display()[DISPLAY]; ... }
  class Room extends Entity
  { Room(); Item item(int n); ... }
  class Door extends Entity
  { Door();
    boolean enter(Player p)[DISPLAY]; ... }
  class Player extends Entity
  { Player(String name);
    void exec()[DISPLAY]; ... }
  class Item extends Entity { Item(); ... }
}
```

Figure 16: The package signature mzbbase2.

```
atom opbase2 {
  arg [DISPLAY];
  open maze : empty -> mzbbase2;
}
// file: opbase2/maze/Entity.java
package maze;
public class Entity extends Object
{ public abstract void display(); ... }
// file: opbase2/maze/Player.java
package maze;
public class Player extends Entity {
  public void display() { ... }
  public void exec() {
    ... Room current = ...; current.display();
    ... Door toEnter = ...; toEnter.display(); ...
  }
}
// file: opbase2/maze/Room.java
package maze;
public class Room extends Entity {
  public void display()
  { Entity in[] = ...; in[0].display(); ... }
}
```

Figure 17: The atom opbase2 and the Java source implementation of its open maze game classes Entity, Player, and Room.

display method arguments, which is reflected in mzbbase2.

The atom opbase2 in Figure 17 uses the package signature mzbbase2 to describe the open package maze. The [DISPLAY] argument parameter from mzbbase2 in atom opbase2 is unbound, which allows the signature of opbase2 to be redefined when it is linked with other units. In the Java source implementation of atom opbase2, [DISPLAY] is invisible: the display method appears as if it has no arguments at all.

One significant restriction is placed on methods whose declarations are modified with an unbound [DISPLAY] argument parameter: the methods can only be called by other methods whose declarations are also modified by [DISPLAY]. As a result, only definitions of the methods display, enter, and exec may call each other. This restriction ensures that the new arguments bound to [DISPLAY] are available to calls of these methods. It also mimics traversal method structure, where definitions of the same method declaration are recursively called. Before a traversal method entry-point can be called, e.g., the method exec in class Player, [DISPLAY] must be bound so

```
signature mzcli = mzbbase2 + {
  use package java.io;
  [DISPLAY] = (PrintStream out, int indent);
}
atom opcli {
  open maze : mzbbase2 -> mzcli;
}
signature mzgui = mzbbase2 + {
  use package java.awt;
  [DISPLAY] = (Graphics g);
  class Entity += { Component widget(); }
}
atom opgui {
  open maze : mzbbase2 -> mzgui;
}
```

Figure 18: The package signatures mzcli and mzgui and the atoms opcli and opgui; the use package clause inserts the classes of the specified package into the package signature's namespace.

```
// file: opcli/maze/Player.java
package maze; ...
public class Player extends Entity {
  public void display(PrintStream out,
    int indent)
  { ... super.display(out,indent); ...
    out.print(this.name()); ... }
  public void exec(PrintStream out, int indent)
  { out.print('Begin'); super.exec(out, 1); }
}
// file: opcli/maze/Room.java
package maze; ...
public class Room extends Entity {
  public void display(PrintStream out,
    int indent)
  { ... super.display(out, indent + 1); ... }
}
```

Figure 19: The Java source code for open maze game classes Player and Room in the implementation of atom opcli.

all display method arguments are known.

The [DISPLAY] argument parameter is bound to fixed argument lists when the package signature mzbbase2 is composed in the package signatures mzcli and mzgui of Figure 18. In package signature mzcli, [DISPLAY] is bound to the new arguments of a print stream (from Java's IO library) and an integer indent level. In package signature mzgui, [DISPLAY] is bound to the new argument of a graphics context (from Java's AWT library). The atoms opcli and opgui in Figure 18 use open maze packages with exported structure describe by package signatures mzcli and mzgui to add implementations of the cli and gui features to the maze game classes.

By using the package signatures mzcli and mzgui to describe the maze open packages, new arguments bound to argument parameter [DISPLAY] in those package signatures are visible within the Java source code implementations of atoms opcli and opgui. Shown in Figure 19, the Java source code for atom opcli can "see" the print stream and indent level arguments added by the binding in package signature mzcli. These arguments are added to the method declarations in imported maze structure, and super class calls to the display methods must provide values for these arguments.

```

atom clidriver {
  import maze : mzcli;
  export main : program;
}
// file: clidriver/main/Main.java
package main;
public class Main extends Object {
  public static void main(String args[])
  { maze.Player p = new maze.Player(args[0]);
    ... p.exec(System.out, 0); ... }
}
compound game3 {
  export main : program;
  export maze : mzcli;
  use package java.io;
  link unit opbase2 where
  [DISPLAY] = (PrintStream out, int indent),
  opcli, clidriver;
}

```

Figure 20: The atom `clidriver`, the Java source code for the implementation of `clidriver`, and the compound `game3`.

The atom `clidriver` in Figure 20 implements an application driver for the maze game in the context of *cli* feature. Because the `[DISPLAY]` argument parameter is bound to a fixed argument list in the package signature `mzcli`, the method `main` in the Java source code of atom `clidriver` shown in Figure 20 can call the method `exec`, even though it is not also modified by `[DISPLAY]`.

The compound `game3` at the bottom of Figure 20 links atoms `opbase2`, `opcli`, and `clidriver` together. Argument parameters are bound in compounds in the same way that name parameters are. When the atom `opbase2` is linked, its `[DISPLAY]` argument parameter is explicitly bound in a `where` clause. As a result of this linking, rewriting physically adds the new arguments to declarations of, and propagates the new arguments in calls to, the methods modified by the `[DISPLAY]` argument parameter in atom `opbase2`.

5. Using Jiazzi

In the previous sections, we have demonstrated Jiazzi’s features in AOP using an in-depth example. In this section we discuss details necessary for development using Jiazzi.

5.1 Type Checking

Type checking of a unit in Jiazzi occurs in an *internal-stage*, which occurs when the unit is constructed, and an *external-stage*, which occurs when the unit is linked by a compound. The separation of a unit’s internal and external type checking is what enables separate compilation in Jiazzi. The Java source compiler does standard type checking for Java classes in atoms. The linker performs checks during internal-stage type checking to ensure the unit’s exports are connected correctly. During the external-stage type checking of a unit, the linker performs checks to ensure the unit’s imported packages are connected correctly.

Argument parameters require extra internal-stage type checking in atoms to ensure that methods modified by an unbound argument parameter are only called by other methods modi-

fied by the same argument parameter. This type checking is performed by a post-compiler provided in Jiazzi’s implementation. Name and argument parameters require checks when they are assigned during a unit’s external-stage type checking to ensure their assignments do not create any ambiguities in the unit’s signature. For example, if the name parameter `[A]` is assigned to the fixed identifier `FOO`, but a class described in the unit’s signature has both the methods `set[A]` and `setFoo`, then the assignment must be rejected because it creates an ambiguity. Because the linker renames methods and classes of a unit as they are linked into a compound, ambiguities cannot occur unless they are apparent in the signature of a unit.

The primary type checking performed by Jiazzi’s linker is oriented around connections between imported and exported packages. Given a connection between two packages, type checking ensures that the *source package* of the connection correctly provides for the *sink package* of the connection [18]. The definition of connection correctness is mostly straightforward: the source package must provide all that is required by the sink package, including classes, name and argument parameters, subclassing relationships, fields, initializers, and methods. There are two exceptions that prevent source packages from providing “too much.” To detect inheritance cycles, the source package must not provide extra subclassing relationships between classes that are visible in the sink package. To ensure that abstract methods in concrete classes can always be detected and rejected, the source package must not provide any abstract methods that are not required by the sink package.

5.2 Implementation

Jiazzi does not require extensions to the Java language: all of Jiazzi’s features are implemented in the linking language that is used to define package signatures, atoms, and compounds. An atom is implemented with source code written in the conventional Java language. The linker in Jiazzi does not process Java source code; the interface between Java and Jiazzi occurs at the level of Java bytecode. Before the Java source code implementation of an atom can be processed by the linker in Jiazzi, it must be compiled into Java bytecode by a conventional Java source compiler such as `Javac`.

Because conventional Java source compilers do not understand an atom’s imported and open packages, a **stub generator** is provided that examines an atom and the package signatures used to describe its packages, and generates Java bytecode that expose the classes in these packages to Java source compilers and other Java tools that understand Java bytecode but not Jiazzi. To ease the implementation of classes in exported and open packages, the stub generator will also generate skeleton Java source files for classes in these packages if they do not exist already. The generated skeleton file will automatically setup the required open class inheritance relationships, e.g., `class Door extends _super_Door`, and provide skeleton implementations of methods that must be implemented to satisfy the exported structure of the open class.

After the Java source code implementation of an atom is compiled into Java bytecode, it undergoes processing by Jiazzi’s linker. The linker internally links the atom by performing its

portion of first-stage type checking over the atom and then packaging its implementation into its linked form, which is a Java archive (JAR) file of Java bytecode and meta information. A compound is only processed by the linker. The linker performs type checking over all the units linked by the compound. Finally, the linker duplicates the Java bytecode in the linked form of each unit that is linked in the compound, rewrites the bytecode, and coalesces the rewritten Java bytecode into the compound's linked form, which has the same format as an atom's linked form.

How the duplicated Java bytecode of linked units are rewritten in a compound depends on how packages are connected and how name and argument parameters are bound within the compound. The binding of a name parameter will cause the names of classes and methods that embed it to be renamed according to the binding. The binding of an argument parameter causes new arguments to be added to methods modified by it, and a rewriting of modified method implementations to propagate the new arguments to calls of modified methods. Finally, methods, classes, and packages not visible outside of the compound are alpha-renamed so that they are hidden.

Open packages are implemented in Jiazzi using a special inheritance and linking pattern referred to as the *open class pattern*. When implementation is added to an open class, it is added in Java bytecode with a new subclass. However, only an open class's "bottom" subclass is used to create subtyping relationships with other classes, used as types, and instantiated. As a result, the open class can appear as a single class to its clients. A detailed explanation of how the open class pattern works is given in our OOPSLA 2001 paper [17].

6. Related Work

Many languages enable modularization of code for concerns that can be modularized in Jiazzi. Rather Jiazzi supports AOP in the style that it supports concern modularization using a linking metaphor and its focus on principled separate compilation. Jiazzi's support for AOP is similar to Hyper/J [20], where atoms are like hyperslices and compounds are like hypermodules. Linking in Jiazzi and composition in Hyper/J serve the same purpose by giving the programmer explicit control over aspect inclusion and configuration. Aspectual components [15] are also adept at modularizing object crosscutting concerns and, like Jiazzi, emphasize modularity to enable separate reasoning about concern implementations. Others have explored the relationship between units and aspects [6]; the features described in this paper focus concerns that cross-cut objects.

AspectJ [12] modularizes concerns using a weaving metaphor, where concern implementations, known as aspects, are woven into well-defined points of code modules. AspectJ allows meta-programs to directly access the internals of a code module while Jiazzi only supports access to code modules through well-defined signatures. AspectJ does not support separate compilation; aspects and the code fragment they are woven into are compiled at the same time. Additionally, while Jiazzi is only adept at modularizing object crosscutting concerns, AspectJ can modularize a more general class of concerns with its emphasis on aspect weaving.

Open classes in Jiazzi are based on mixins [1], where extensibility is gained with classes whose super classes are initially unfixed. Role-model components [23] use individual mixins to modularize the many "roles" an object is involved in. Mixin layers [22] improve on this by using a layer of mixins to modularize a collaboration between many objects. Java Layers [4, 3] adds mixin layers to the Java language. The *sibling class pattern* [3] enabled by Java Layers provides the same functionality as Jiazzi's open packages. Delegation layers [21] improves on mixin layers by allowing new collaborations to be added at run time.

MultiJava [5] adds direct support for open classes with an extension to the Java language. As in Jiazzi, open classes in MultiJava are fully supported with principled separate compilation. MultiJava open classes are more flexible in that new methods can be added to a class at run time after program execution has begun, while Jiazzi only supports addition to an open class during program linking. Conversely, Jiazzi supports adding new fields to an open class, where MultiJava only supports the addition of new methods.

The name parameter construct in Jiazzi enables the renaming of classes and methods. Renaming is also supported by Hyper/J but not at the sub-identifier level. Programmers commonly use conventions for naming methods and classes, and name parameters allow these conventions to be codified by units. AspectJ also takes advantage of naming conventions in wild-carded point-cut definitions that identify where in a code fragment code should be added. Name parameters also provide a simple way to perform the explicit renaming or name resolution that is useful whenever an OO language supports a forms of multiple inheritance, such as mixins [9].

The argument parameter construct enables context required by method definitions to be encapsulated across concern implementations. An alternative approach is to provide in-language support for variables whose bindings are specified over a dynamic scope. Dynamic scoping is supported in some Lisp-like languages as well as in many domain specific languages like TeX and PostScript. Calls to methods affected by argument parameters are restricted in Jiazzi so the linker can add and pass new method arguments automatically. Rather than use such restrictions, an implicit parameters [14] extension to Haskell uses inference to determine which functions are affected by the implicit parameters. Dynamically-scoped variables have also been proposed as an extension to C# [11].

7. Conclusions and Future Work

Jiazzi supports expressive aspect-oriented programming with units that enable open classes and signatures. By supporting separate compilation, Jiazzi enables separate reasoning of concern implementations that makes their modularization more complete.

While Jiazzi is adept at modularizing concerns whose implementations cleanly crosscut object boundaries, Jiazzi cannot effectively modularize other concerns whose implementations cross-cut the statements and expressions of method definitions. These concerns are best modularized using code weaving mech-

anisms, such as AspectJ which severely complicates separate compilation. Our future work will focus on modularizing these concerns without sacrificing support for separate compilation.

Jiazzi is very pragmatic: it does not modify the syntax of the core Java language and creates binaries that can execute in a Java virtual machine. An implementation of Jiazzi is available for download, and we are currently preparing a new release and tutorial that focuses on the AOP-centric features presented in this paper. For more information, see the Jiazzi website: <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] L. Cardelli. Program fragments, linking and modularization. In *Proc. of POPL*, pages 266–277, Jan. 1997.
- [3] R. Cardone, A. Brown, S. McDirmid, and C. Lin. Using mixins to build flexible widgets. In *Proc. of AOSD*, June 2002.
- [4] R. Cardone and C. Lin. Comparing frameworks and layered refinement. In *Proc. of ICSE*, pages 285–294, May 2001.
- [5] 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.
- [6] E. Eide, A. Reid, M. Flatt, and J. Lepreau. Aspect weaving as component knitting: Separating concerns with knit. In *Workshop on Advanced Separation of Concerns in Software Engineering*, May 2001.
- [7] R. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proc. of ICFP*, pages 98–104, Sept. 1998.
- [8] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proc. of PLDI*, pages 236–248, May 1998.
- [9] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. of POPL*, pages 171–183, Jan. 1999.
- [10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [11] D. R. Hanson and T. A. Proebsting. Dynamic variables. In *Proc. of PLDI*, May 2000.
- [12] G. Kiczales, E. Hilsdale, J. Hungunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proc. of ECOOP*, June 2001.
- [13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of ECOOP*, June 1997.
- [14] J. R. Lewis, M. B. Shields, E. Meijer, and J. Launchbury. Implicit parameters: Dynamic scoping with static types. In *Proc. of POPL*, pages 108–118, Jan. 2000.
- [15] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with aspectual components. 1999.
- [16] S. McDirmid. *The Jiazzi Manual*, 2002.
- [17] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proc. of OOPSLA*, Oct. 2001.
- [18] S. McDirmid, M. Flatt, and W. C. Hsieh. Expressive modular linking for object-oriented languages. Technical Report UUCS-02-014, 2002.
- [19] T. Millstein and C. Chambers. Modular statically typed multimethods. In *Proc. of ECOOP*, pages 279–303, July 1999.
- [20] H. Ossher and P. Tarr. Hyper/J: multi-dimensional separation of concerns for Java. In *Proc. of ICSE*, pages 734–737, June 2000.
- [21] K. Ostermann. Dynamically composable collaborations with delegation layers. 2002.
- [22] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proc. of ECOOP*, pages 550–570, June 1998.
- [23] M. VanHilst and D. Notkin. Using role components to implement collaboration-based designs. In *Proc. of OOPSLA*, pages 359–369, Oct. 1996.