**Summary:** Jiazzi is a language for defining and linking program components written in the Java language. Jiazzi components contain ordinary Java classes, which may be exported from components for import in other components. A component can use imported classes normally; they can be instantiated or subclassed.

The actual class for a component’s import is not determined until component link time. Nevertheless, Jiazzi supports separate compilation (and type checking) of components and type-safe linking. To maximize flexibility, Jiazzi allows mutually recursive components, and allows the actual class supplied for a component’s import to provide more methods than the component expects. Potential problems with cyclic inheritance graphs and colliding method definitions are addressed by the linking language.

Jiazzi is the first combination of components and classes to support the flexibility of both mutual linking dependencies and imports that contain unexpected methods. The flexibility provided by Jiazzi allows programmers to create generic collection classes and mixin-like constructions, and provides a solid foundation for building class frameworks.
1. Introduction

Current Java constructs for reuse—including classes, interfaces, and packages—are insufficient for implementing reusable components [32]. In addition to classes, which enable code reuse at the level of object implementations, Java needs constructs for building large-scale subsystems out of separately compiled, externally linked components [31].

A practical component system for Java must meet several criteria:

• It must allow classes to be imported and exported across component boundaries, and allow a component to both instantiate and subclass an imported class.

• It must enable separate compilation (and type-checking) of components to support the development of large programs and the distribution of components in compiled form.

• It must support components that specify only the shapes of their imports, as opposed to designating specific components that supply the imports. Linking among components is specified by client programmers (programmers who assemble components) in a separate, external language, which gives them maximal flexibility in composing programs out of components.

• It must permit mutual dependencies (linked imports and exports) among components. Allowing cycles in the import-dependency graph is particularly important at the level of classes, where mutually-recursive “has a” relationships are common.

• It must allow a class provided for a component’s import to supply more methods than the component’s use of the import requires. Requiring an exact match on the methods of an imported class would prohibit useful composition of class-extending components (a.k.a. mixins [5][14]).

Jiazzi, our new component language for Java, provides the first combination of components and classes that satisfies all of the above requirements. The last two requirements, in particular, each pose an interesting technical problem:

(1) Mutually-recursive linking and subclassing of classes across component boundaries can introduce circular inheritance hierarchies. These must be detected and rejected at component-link time.

(2) Allowing imported classes to provide more methods than expected by a component creates the possibility of method collisions: a component might import a class C that adds a method m in a subclass, but the actual class supplied for C might unexpectedly provide a method m. Method collisions must be either resolved automatically or rejected.

Jiazzi addresses the first problem by including enough information in component signatures to support inheritance-cycle detection. This signature information derives from the component definition in a straightforward way, and allows for separate type checking and compilation. Jiazzi addresses the second problem by rejecting component compositions that introduce method conflicts, but provides method-scoping rules that enable programmers to avoid many kinds of collisions.

Many of the techniques used to define Jiazzi have been explored previously: the core component model is derived from program units [14][17], and Jiazzi’s hiding of private methods resembles that of Riecke and Stone [28] and Vouillon [33]. Our contribution in Jiazzi is demonstrating how these techniques can be combined to define a practical component system for Java and related languages.

Since Jiazzi is intended as a practical tool, we have avoided design choices that would require changing the core Java language, the existing runtime infrastructure, or the way that programmers reason about the behavior of Java source code. Instead, Jiazzi acts as a component-configuration language outside of normal Java code.

In this paper, we describe Jiazzi language, and a small model of the language that illustrates how components can be safely linked. Section 2 gives an overview of Jiazzi and specific examples of how its features can be used. Section 3 gives a formal description of the core Jiazzi semantics, and Section 4 describes the implementation of Jiazzi. Section 5 discusses related work, and Section 6 summarizes our conclusions.

2. Overview

Jiazzi, our new component language for Java, provides the first combination of components and classes that satisfies all of the above requirements. The last two requirements, in particular, each pose an interesting technical problem:

(1) Mutually-recursive linking and subclassing of classes across component boundaries can introduce circular inheritance hierarchies. These must be detected and rejected at component-link time.

(2) Allowing imported classes to provide more methods than expected by a component creates the possibility of method collisions: a component might import a class C that adds a method m in a subclass, but the actual class supplied for C might unexpectedly provide a method m. Method collisions must be either resolved automatically or rejected.

Jiazzi addresses the first problem by including enough information in component signatures to support inheritance-cycle detection. This signature information derives from the component definition in a straightforward way, and allows for separate type checking and compilation. Jiazzi addresses the second problem by rejecting component compositions that introduce method conflicts, but provides method-scoping rules that enable programmers to avoid many kinds of collisions.

Many of the techniques used to define Jiazzi have been explored previously: the core component model is derived from program units [14][17], and Jiazzi’s hiding of private methods resembles that of Riecke and Stone [28] and Vouillon [33]. Our contribution in Jiazzi is demonstrating how these techniques can be combined to define a practical component system for Java and related languages.

Since Jiazzi is intended as a practical tool, we have avoided design choices that would require changing the core Java language, the existing runtime infrastructure, or the way that programmers reason about the behavior of Java source code. Instead, Jiazzi acts as a component-configuration language outside of normal Java code.

In this paper, we describe Jiazzi language, and a small model of the language that illustrates how components can be safely linked. Section 2 gives an overview of Jiazzi and specific examples of how its features can be used. Section 3 gives a formal description of the core Jiazzi semantics, and Section 4 describes the implementation of Jiazzi. Section 5 discusses related work, and Section 6 summarizes our conclusions.

2. Overview

A Jiazzi program consists of a set of component definitions known as units. A unit is either an atom, which has associated class definitions, or a compound, which does not have class definitions directly but instead links other units together.
Informally, a compound can be reduced to an atom by:

(1) Taking the (reduced) bodies of all units listed in the make section, and creating one copy for each local-unit-name instance.

(2) Concatenating all of the copied bodies, and renaming classes according to the mapping specified by the link section. This rewriting is similar to how a linker finds and updates offsets at link time.

The result is a unit that can be either loaded and executed (if it has no imports), or linked to form larger compounds.

Of course, there is no guarantee that the concatenated bodies make sense unless some form of checking has been applied to units and unit linking along the way. Informally, such type-checking rules are most easily explained through examples. We provide a formal description of type checking in Section 3.

A unit can be executed when it has no imports. In that case, the execution of the unit is the same as executing the Java code contained within the unit, ignoring the unit exports (if any). However, if a unit has imports, it can only be used in a compound for linking.

In the shape of a class, if the class-name after extends is not Object, it must be associated with some other shape in the unit’s imports and exports. It cannot be an unbound name or the name of an unexported class defined within the unit. This restriction also applies to any class-name that appears in the shape’s field-and-methods.

The field-and-methods in a shape are analogous to the field and method prototypes that appear in a C++ class declaration. The actual implementation of the methods will come from some atom’s class-definition which corresponds to a set of Java class files.

A unit can be executed when it has no imports. In that case, the execution of the unit is the same as executing the Java code contained within the unit, ignoring the unit exports (if any). However, if a unit has imports, it can only be used in a compound for linking.

Figure 1: PROVIDE-BAG provides a parameterized collection class.

Figure 2, the compound LINK-CLIENT-TO-BAG instantiates the units PROVIDE-BAG and PROVIDE-CLIENT to create bag and client, respectively. The import G of the compound is linked to F of bag and E of client. BagOfE, exported by bag, is linked to the BagOfF import of client. Finally, the exported class FClient from client is exported from the compound as GClient. This linking is illustrated in Diagram 1.

Only local unit signatures are used to type-check linking inside a compound. These signatures consist of the shapes of local unit imports and exports. In these shapes, references to formal imports of local units are replaced with references to the actual classes linked to them. In Figure 2, BagOfE in local unit bag can be linked to BagOfF in local unit client because the same class G is linked to both E in bag and F in client. Using signatures, type checking a link operation does not require implementation knowledge of the local unit classes, so units can be type-checked separately.

```plaintext
imports ::= (import shape)*
exports ::= (export shape)*
shape ::= class class-name extends class-name {
    field-and-methods
}

In the shape of a class, if the class-name after extends is not Object, it must be associated with some other shape in the unit’s imports and exports. It cannot be an unbound name or the name of an unexported class defined within the unit. This restriction also applies to any class-name that appears in the shape’s field-and-methods.

The field-and-methods in a shape are analogous to the field and method prototypes that appear in a C++ class declaration. The actual implementation of the methods will come from some atom’s class-definition which corresponds to a set of Java class files.

A unit can be executed when it has no imports. In that case, the execution of the unit is the same as executing the Java code contained within the unit, ignoring the unit exports (if any). However, if a unit has imports, it can only be used in a compound for linking.

```

```plaintext
linkage ::= make link
make ::= make (local-unit-name as unit-name)*;
link ::= link (local-out to compound-out | local-in to local-in | compound-in to local-in)*;

The linkage in a compound is divided into two sections. In the make section, each unit-name specifies a unit to be linked in the compound unit. Since a single unit can be used more than once in a single compound, each use of the unit gets its own local-unit-name to be used in the link section.

The link section connects the imported and exported classes to each other. Each local-[out/in] is of the form local-unit-name.class-name, where class-name is either an export (in the case of local-out) or an import (in the case of local-in) of the local unit specified by local-unit-name. Each compound-[out/in] is of the form class-name, where class-name is either an export (in the case of compound-out) or an import (in the case of compound-in) of the compound.

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

```
Jiazzi units can derive classes that behave much like mixins [5] by exporting classes derived from imports. Such an export will have any methods present in the actual class linked to its superclass. If a method \( m \) is not described in the import’s shape, then it is hidden from classes inside the local unit. However, any export that subclasses the import will contain \( m \) when viewed in the enclosing compound.

In Figure 4, color is added to both the compound import Point using the unit MIXIN-COLOR by linking Point to \( E \) in \( mix \). The export of \( mix \), ColorE, can then be linked to the import ColorPoint of the local unit \( use \). This linking is possible since methods \( \text{getX}, \text{getY} \) are provided by Point and the methods \( \text{getColor} \) and \( \text{setColor} \) are provided by ColorE of \( mix \).

2.1. Mixins

Jiazzi units can derive classes that behave much like mixins [5] by exporting classes derived from imports. Such an export will have any methods present in the actual class linked to its superclass. If a method \( m \) is not described in the import’s shape, then it is hidden from classes inside the local unit. However, any export that subclasses the import will contain \( m \) when viewed in the enclosing compound.

In Figure 4, color is added to both the compound import Point using the unit MIXIN-COLOR by linking Point to \( E \) in \( mix \). The export of \( mix \), ColorE, can then be linked to the import ColorPoint of the local unit \( use \). This linking is possible since methods \( \text{getX}, \text{getY} \) are provided by Point and the methods \( \text{getColor} \) and \( \text{setColor} \) are provided by ColorE of \( mix \).

Jiazzi requires that maximal subtyping information is always visible in a unit. If a \( C \) is not visibly a subtype of \( C' \) in a scope where both \( C \) and \( C' \) are visible, then \( C \) is not allowed to be a subtype of \( C' \) in any scope. This restriction prevents hiding of inheritance relationships between an export and other exports and imports. It also prevents the hiding of inheritance relationships between actual imports by the formal imports of a local unit. By maintaining the maximal-subtype invariant in a unit, link-time type checking can detect class circularity without sacrificing the ability to check units separately. Maximal-subtype invariant also eliminates overloading ambiguity; the methods chosen for call sites at compile time will remain the correct choices after link time. Finally, having maximal-subtyping information allows

```
atom PROVIDE-CLIENT {
    import class F { int hashCode(); }
    import class BagOFF {
        void add(F f); boolean contains(F f);
    }
    export class FClient {
        void run(F f);
    }
}

compound LINK-CLIENT-TO-BAG {
    import class G { int hashCode(); }
    export class GClient { void run(G g); }
    make bag as PROVIDE-BAG,
    client as PROVIDE-CLIENT;
    link G to bag.E,
    G to client.F,
    bag.BagOfE to client.BagOff,
    client.FClient to GClient;
}

Figure 2: The collection in Figure 1 is used to create a bag of \( G \), which is then used in another local unit.

Diagram 1: This is an illustration of the unit LINK-CLIENT-TO-BAG in Figure 2. Imports enter a unit from the top and exports exit the unit from the bottom.

atom TREE {
    import class BagOfG {
        void add(G g); boolean contains(G g);
    }
    export class G {
        int hashCode(); BagOfG children();
    }
}

compound LINK-TREE {
    make bag as PROVIDE-BAG,
    tree as TREE;
    link tree.G to bag.E,
    bag.BagOfE to tree.BagOfG;
}

Figure 3: The unit TREE exports a class \( G \) that implements a recursive tree structure.

Mutually dependent linking necessarily arises when the shape of an import refers to exports. In Figure 3, local units \( bag \) and \( tree \) are linked to each other in the compound LINK-TREE, which is necessary since \( G \) in \( tree \) “has a” bag of itself. As a result, \( G \) is linked to \( E \) in \( bag \), while \( BagOfE \) in \( bag \) is linked to \( BagOfG \) in \( tree \).
all runtime coercion behavior to remain locally obvious inside a unit.

**Design note:** Mixins in Jiazzi create class definitions only; they do not provide a common type to describe the functionality they add, unlike other mixin proposals such as JAM [1]. By ensuring that mixin instances have separate types, we simplify the semantics, avoid overloading that causes method collisions, and allow mixins to be applied more than once to the same class. However, having a common type for mixin applications can be useful (e.g., to recognize all colored types created using MIX-COLOR). We are currently investigating using Java interfaces to provide a solution for this problem.

### 2.2. Constraining Method Scopes

In Java, if a class \( C \) declares a method \( m \) and its superclass \( C' \) also declares \( m \), then the declaration of \( m \) in \( C \) always overrides the one in \( C' \). In Jiazzi, however, the declaration of \( m \) in \( C \) does not always override the superclass declaration if \( C' \) is imported. To maintain modularity, overriding only occurs if the shape for \( C' \) contains \( m \).

Figure 5 shows a case where overriding does not occur, despite the fact that a class (**Cowboy**) and its superclass (**Icon**) both contain a definition for a particular method (draw). Since the shape of the **Super** import for **COWBOY-UNIT** does not mention the draw method, the implementor of the **Cowboy** class should not expect a draw method to exist in **Super**. Consequently, the draw method declared in the **Cowboy** class must be a method introduction and not a method override. However, in the linking of classes within **COWBOYICON-UNIT**, it turns out that **Icon**, the class supplied for **COWBOY-UNIT**'s **Super**, does have a draw method. Nevertheless, the two draw methods are kept separate: within **COWBOY-UNIT**, draw refers to the gun-slinging draw; within **ICON-UNIT** and outside of **COWBOYICON-UNIT**, draw refers to the graphical draw operation on icons.

```plaintext
atom ICON-UNIT {
  import class Super {}
  export class Icon extends Super {
    void draw();
  }
}
atom COWBOY-UNIT {
  import class Super {}
  export class Cowboy extends Super {
    void duel();
  }
}
compound COWBOYICON-UNIT {
  export class CowboyIcon {
    void draw(); void duel();
  }
  make icon as ICON-UNIT, cowboy as COWBOY-UNIT;
  link icon.Icon to cowboy.Super,
    cowboy.Cowboy to CowboyIcon;
  }
  /* code for Cowboy in COWBOY-UNIT */
*/
  class Cowboy extends Super {
    void draw() { ... }  
    void duel() { ... draw(); ... }
  }
}
Figure 5: The unit COWBOYICON-UNIT links creates a class that is both a cowboy and an icon.

Figure 5 illustrates how, from a global perspective, a single class can contain multiple methods with the same name; within each local scope there is no ambiguity. However, if the **Cowboy** class exported from **COWBOY-UNIT** exposes its draw method, as shown in Figure 6, then the draw method exported by **COWBOYICON-UNIT** would be ambiguous. In that case, Jiazzi would reject the linking of **ICON-UNIT** with **COWBOY-UNIT**.

```plaintext
atom COWBOY-UNIT {
  import class Super {}
  export class Cowboy extends Super {
    void duel();
  }
}
``` Figure 6: This revised version of COWBOY-UNIT exposes its draw method. It cannot be linked using COWBOYICON-UNIT in Figure 5 because a class cannot visibly have two draw methods.
atom NEEDS-KEY {
    import class Door () import class Key {}
    export class KeyDoor extends Door {
        void setKey(Key k); Key getKey();
        Object neededPass();
    }
}

atom NEEDS-SPELL {
    import class Door () import class Spell {}
    export class SpellDoor extends Door {
        void setSpell(Spell s); Spell getSpell();
        Object neededPass();
    }
}

atom SECURE {
    import class Door {
        boolean canOpen(Person p);
        Object neededPass();
    }
    import class Person {
        boolean hasItem(Object o);
    }
    export class SecureDoor extends Door {
    }
    /* Implementation of SecureDoor */
    class SecureDoor extends Door {
        boolean canOpen(Person p) {
            if ((p.hasItem(this neededPass())))
                return false;
            else return super.canOpen(p);
        }
    }
    Figure 7: The units NEEDS-KEY and NEEDS-SPELL enhance doors to provide the neededPass method needed by the unit SECURE. The unit SECURE updates door to check if a needed item is possessed by a person in a maze game.

    Figure 7–Figure 10 illustrate how limiting method scopes works in a more sophisticated example. The units in Figure 7 enhance doors in an adventure game. Local units created from the unit SECURE create a new type of door that prevents a person from entering it without a specific object. Instances of the units NEEDS-KEY and NEEDS-SPELL are connected to local units of SECURE so that the object needed is either a Key or a Spell.

    compound LOCKED-MAGIC-INCORRECT {
        import class Door {
            boolean canOpen(Person p);
        }
        import class Person {
            boolean hasItem(Object o);
        }
        import class Key () import class Spell {}
        export class LockedMagicDoor extends Door {
            Key getKey(); void setKey(Key k);
            Spell getSpell(); void setSpell(Spell s);
        }
        make key as NEEDS-KEY, lsec as SECURE,
        spell as NEEDS-SPELL, msec as SECURE;
        link Door to key Door,
        Key to key Key,
        key.KeyDoor to lsec Door,
        Person to lsec Person,
        // next line causes linking error
        lsec.SecureDoor to spell Door,
        Spell to spell Spell,
        magic.SpellDoor to msec Door,
        Person to msec Person,
        msec.SecureDoor to LockedMagicDoor;
    }
    Figure 8: The compound LOCKED-MAGIC-INCORRECT incorrectly links of instances of atoms SECURE, NEEDS-KEY, and NEEDS-SPELL to form magic locked doors.

    compound SECURE-LOCKED {
        import class Door {
            boolean canOpen(Person p);
        }
        import class Person {
            boolean hasItem(Object o);
        }
        import class Key ()
        export class LockedDoor extends Door {
            void setKey(Key k); Key getKey();
        }
        make key as NEEDS-KEY, secure as SECURE;
        link Door to key Door,
        Key to key.Key,
        key.KeyDoor to secure Door,
        Person to secure Person,
        secure.SecureDoor to LockedDoor;
    }
    compound SECURE-MAGIC {
        import class Door {
            boolean canOpen(Person p);
        }
        import class Person {
            boolean hasItem(Object o);
        }
        import class Spell {}
        export class MagicDoor extends Door {
            void setSpell(Spell s); Spell getSpell();
        }
        make spell as NEEDS-SPELL, secure as SECURE;
        link Door to spell Door,
        Spell to spell Spell,
        spell.SpellDoor to secure Door,
        Person to secure Person,
        secure.SecureDoor to MagicDoor;
    }
    Figure 9: The compounds SECURE-LOCKED and SECURE-MAGIC limits the scope of neededPass.

    Figure 8 illustrates an incorrect linking of the units in Figure 7. Each magic and locked part of door requires
unique behavior and state, so **Secure** is instantiated twice. The unit in Figure 8 is not a well-formed compound because a method collision occurs when creating magic locked doors. The local unit *spell* adds the method *neededPass* to its **SpellDoor** export. The same method is already visible in the superclass **KeyDoor** of the local unit **key**. Therefore, the linking in Figure 8 fails; Jiazzi prohibits the linking of local units together in a way that causes method collisions.

A client programmer can fix the method collision in Figure 8 by creating new enclosing compounds to limit the scope of *neededPass*, shown in Figure 9. This solution is possible because *neededPass* is only used by the local units of **Secure**. These compounds can then be used in Figure 10 to create a locked magic door. Diagram 2 illustrates the linking structure in Figure 10.

**Design note:** It can sometimes be useful to have colliding methods be visible (e.g., both key and spell *neededPass* methods may be needed at the same time), and let the client programmer choose which one to call. In Jiazzi, method collisions must be resolved in favor of one method only. Moby [16], in contrast, allows method collisions, and leaves the complexity of resolving collisions to the caller. We are currently investigating hybrid mechanisms that would provide more flexibility in managing colliding methods.

```java
compound LOCKED-MAGIC-CORRECT {
  import class Door {
    boolean canOpen(Person p);
  }
  import class Person {
    boolean hasPass(Object item);
  }
  import class Key {} import class Spell {}
  export class LockedMagicDoor extends Door {
    void setKey(Key k); Key getKey();
    void setSpell(Spell s); Spell getSpell();
  }
  make locked as Secure-Locked,
    magic as Secure-Magic;
  link Door to locked.Door,
    Person to locked.Person,
    Key to locked.Key,
    locked.LockedDoor to magic.Door,
    Person to magic.Person,
    Spell to magic.Spell,
    magic.MagicDoor to LockedMagicDoor;
}
```

**Figure 10:** Using the compounds from Figure 9, we can create a correct linking of a locked magic door.

### 2.3. Reverse Mixin Pattern

In the **reverse mixin pattern**, an import subclasses an export in a unit. Because of the subclassing relationship, any details about the exported class hidden inside the unit remain visible in objects of the imported class. The reverse mixin pattern exploits mutually dependent linking of local units to allow a class to be implemented across multiple units, similar to the functionality provided by Duggan and Sourelis’s mixin modules [12].

Using the reverse mixin pattern, two imports, C<sub>Original</sub> and C<sub>Final</sub>, and one export, C<sub>Out</sub>, are used to represent a single class inside a unit. The export C<sub>Out</sub> adds implementation. The import C<sub>Original</sub>, which C<sub>Out</sub> subclasses, represents the class’s original implementation. The import C<sub>Final</sub>, which subclasses C<sub>Out</sub>, represents the final implementation of the class. With correct usage of the pattern, C<sub>Final</sub> is the only representative of the class that is instantiated, subclassed by classes, and used in the type signatures of fields. In many ways, C<sub>Final</sub> is similar to a virtual type [25][8], or even a self-type [7], though the actual type is determined at link time, not runtime.

Figure 11 demonstrates how a reverse mixin pattern can be used in a design. In the APPLY-DUPL-OOK unit, double link functionality is added to a class F<sub>In</sub>. This functionality is added by both the units SINGLE-LINK and DOUBLE-LINK. Even though SINGLE-LINK only implements single link functionality, calling the method...
getNext in and outside of the compound will return an instance of \( F_{\text{out}} \) that has double link functionality. The unit RAISE is necessary to avoid linking Double-Link’s \( E_{\text{out}} \) directly to its \( E_{\text{final}} \). The linking is illustrated in Diagram 3.

Diagram 3: This is an illustration of the linking done in Figure 11.

atom SINGLE-LINK { 
  import class \( E_{\text{orig}} \) 
  import class \( E_{\text{final}} \) extends \( E_{\text{out}} \) 
  export class \( E_{\text{out}} \) extends \( E_{\text{orig}} \) { 
    void setNext(\( E_{\text{final}} \) e); \( E_{\text{final}} \) getNext(); 
  }
}

atom DOUBLE-LINK { 
  import class \( E_{\text{in}} \) 
  void setNext(\( E_{\text{final}} \) e); \( E_{\text{final}} \) getNext(); 
}

atom RAISE { 
  import class In 
  export class Final extends In [] 
}

compound APPLYDOUBLE-LINK { 
  import class \( F_{\text{in}} \) 
  export class \( F_{\text{out}} \) extends \( F_{\text{in}} \) { 
    \( F_{\text{out}} \) getNext(); \( F_{\text{out}} \) getPrevious(); 
    void setNext(\( F_{\text{out}} \) f); 
    void setPrevious(\( F_{\text{out}} \) f); 
  }
}

make single as SINGLE-LINK, 
  double as DOUBLE-LINK, 
  raise as RAISE; 
link \( F_{\text{in}} \) to single.\( E_{\text{orig}} \), 
  single.\( E_{\text{out}} \) to double.\( E_{\text{orig}} \), 
  double.\( E_{\text{out}} \) to raise.In, 
  raise.\( E_{\text{final}} \) to single.\( E_{\text{final}} \), 
  raise.\( E_{\text{final}} \) to double.\( E_{\text{final}} \), 
  raise.\( E_{\text{final}} \) to \( F_{\text{out}} \); 

Figure 11: SINGLE-LINK enhances its original class with built in single link list functionality. DOUBLE-LINK builds on SINGLE-LINK functionality to enhance a class with a backwards link. APPLYDOUBLE-LINK uses the reverse mixin pattern to enhance a generic class \( F_{\text{in}} \) with double linked list functionality.

Design note: In Jiazzi, an export of a local unit can never be linked to an import in the same local unit. Allowing this would violate the maximal subtyping invariant and allow overloading that could cause method collisions at link time. Consequently, the local unit that adds the complete implementation of a class cannot directly use its export to satisfy its import of the class. However, this problem has an easy solution. We use a unit whose only purpose is to create a single export that is a subclass of a single import while adding no implementation. This export can then be linked to the \( C_{\text{final}} \) imports of the class in the pattern. In Figure 11 the RAISE unit serves this purpose.

Design note: Using the reverse mixin pattern in Jiazzi does not make this in the implementation of \( C_{\text{out}} \) a true self-type. By convention of the pattern, only the class linked to the import \( C_{\text{final}} \) is instantiated. The runtime type of this should always be that of the \( C_{\text{final}} \). However, since the pattern cannot be enforced in either the Java or Jiazzi language, the static type of this remains that of \( C_{\text{out}} \). When this is reflexively used as an argument to a binary method, it must be explicitly coerced to the type of \( C_{\text{final}} \), though the cast will not fail at runtime if the pattern is adhered to.

The reverse mixin pattern can be used to implement class frameworks. A framework is an abstract object-oriented design composed of tightly related classes [22] that are specialized together. In Jiazzi, we can implement a framework in layers of units using the reverse mixin pattern. We are currently exploring adding direct support for class frameworks based on our experiences using the reverse mixin pattern.

3. MiniJiazzi

To illustrate how Jiazzi maintains type safety while linking components, we formally study a simplified version of the language, MiniJiazzi.

Notation: An overbar indicates a sequence or set, depending on context. For example, \( \overline{\text{shape}} \) indicates a sequence or set of \( \text{shape} \), and \( \text{unit}(C) \) indicates a sequence of \( \text{unit} \), each with its own sequence of \( C \)’s.

A cross superscript (\( \times \)) on a relation maps the relation over a sequence of operands to generate a new sequence; a cross superscript on the left maps the relation over its left operand, and a cross superscript on the right maps the relation over its right operand. For example, \( \text{shape}_{i} \overline{\text{shape}} \) applies the intro judgment to each individual imported shapes with respect to the to a sequence of signature shapes. \( \text{shape}_{i} \overline{\text{shape}} \) applies the \text{NOTNEARERSUPER} rule to each individual exported shape with respect to each individual signature shape.
∀unit, \vdash \text{unit} : \{ \text{import} \{ \} \text{export} \{ \ldots \text{main extends} \ldots \{ \ldots \text{main}[\ldots]() \ldots \} \ldots \} \}
\Rightarrow \exists c, \text{VALUEOF unit \text{ISA} c}

\text{VALUEOF unit \text{ISA} c} \iff \text{unit} \rightarrow_s \{ \text{import} \text{export} \{ \ldots \text{main extends} \ldots \{ \ldots \text{main}[c]() \ldots \} \ldots \}
\text{body} \text{class} \}
\quad \text{and} \quad \text{class} \vdash \text{(new main).main}[c]() \rightarrow_s \text{new} c

\text{Figure 12: MiniJiazzi soundness}

\begin{align*}
\text{unit} & = \text{atom} \mid \text{compound} \\
\text{atom} & = \text{atom} \{ \text{import} \text{shape} \text{export} \text{shape} \}
\quad \text{body} \text{class} \\
\text{compound} & = \text{compound} \{ \text{import} \text{shape} \text{export} \text{shape} \}
\quad \text{make} \text{unit}(c) \\
\text{shape} & = c \text{extends} c \{ \text{mdecl} \} \\
\text{class} & = c \text{extends} c \{ \text{fresh} \text{meth} \text{override} \text{meth} \} \\
\text{mdecl} & = c \text{m}[c](c') \{ \text{expr} \} \\
\text{meth} & = c \text{m}[c][c'c](c') \{ \text{expr} \} \\
\text{expr} & = \text{new} c \{ \text{expr.m}[c][\text{expr}][x] \}
\quad c = \text{a class name or Object} \\
\quad m = \text{a method name} \\
\quad x = \text{a variable or this}
\end{align*}

\text{Figure 13: MiniJiazzi syntax}

The syntax for MiniJiazzi is shown in Figure 13. Whereas a Jiazzi program contains a sequence of units, a MiniJiazzi program consists of a single unit expression, though it can be a compound expression containing other unit expressions. The top-level unit expression must have no imports, and it must export a \text{Main} class with a \text{main} method that takes no arguments.

A unit expression consists of a sequence of shapes that describe its imports and exports. Following the imports and exports, an \text{atom} expression contains a sequence of class definitions while a \text{compound} expression contains local unit expressions and linking instructions. Unlike a Jiazzi unit, a unit expression is only used in one context and is embedded directly inside a compound expression. To simplify the model, we require that the exports of all local unit expressions be uniquely named. This allows us to use names equivalence between the local unit expression’s export and the compound expression’s export when determining the connection of a compound’s exports. This requirement is a limitation compared to full Jiazzi, where linking labels disambiguate exports among the local units, but it simplifies the semantics.

A class definition names a class, its superclass, introduces fresh method implementations that must not be present in its superclass, and defines overriding implementations of methods already present in its superclass. The shape of a class (in the imports and exports) contains the fresh method declarations and superclass relationship that are visible to the unit’s enclosing compound expression. Like Jiazzi, the shape of an import or export can refer only to other imports and exports of the unit expression.

Method declarations and definitions have the form \text{m}[c], which identifies both the method and the source class for the method (i.e., the class where the method is declared as a fresh method). A method declared in a shape or a fresh definition always names the containing shape or class, respectively, as the source of the method. In the case of an imported shape, this source will be automatically refined during linking, depending on the actual class supplied to the unit. In contrast to Jiazzi, two method declarations visible in a class cannot have the same name in MiniJiazzi (no overloading). A method call in an expression specifies only the name of the method, not its source class.

The root class name \text{Object} and the special variable \text{this} are built into MiniJiazzi. We assume that \text{Object} and \text{this} are never bound by the program, that all class and variable bindings within an atom are unique, and that all imported and exported class names in a compound are unique. A MiniJiazzi class contains no fields, so an instance of class \text{c} can be represented simply by \text{new} \text{c}. The only reducible expression form is a method call.

The soundness of MiniJiazzi, as expressed in Figure 12, ensures that method calls will always succeed when calling the \text{main} method of the \text{Main} class in the top-level unit expression. The soundness claim relies on two reductions, a linking reduction (→_n) and an expression reduction (→_e), along with multiple judgments (→). The expression reduction and expression-level type judgments are essentially the same as in previous work [17], so we have omitted them. The unit-level type judgments and linking reduction are described in the following sections.

3.1. Type Checking

At the level of unit and linking expressions, the type-checking environment consists of a set of shapes that represent classes imported and exported into the unit as well as class definitions inside the unit. The type
judgment (|-_) in Figure 14 shows how subtyping relationships are derived.

The NOTNEARERSUPER rule is used to ensure that an exported class's shape exposes subtyping information of its class definition that is maximal with respect to other imported and exported classes. If \( e \) \textit{extends} \( c' \) is an export's shape, then every imported or exported class \( c'' \) that is a supertype of \( e \) must also be a supertype of \( c' \). Using this rule, cyclic inheritance can be type-checked since the relationship of an exported class with an imported class is always known. This rule is also necessary to prevent some method collisions between imported classes and exported classes.

The primary method judgment (|-\textit{meth}) describes how methods inside a class are extracted from the environment. The SATISFFIES rule is used to ensure that a class supplied for a unit expression's import conforms to the shape of the import.

\[
\begin{align*}
\text{shape} \vdash c & \quad \text{shape} \vdash c' \quad \text{shape} \vdash c < c' \\
\text{shape} \vdash \text{Object} & \quad < \text{is transitively closed} \\
\text{shape} \vdash \text{Object} \leq c & \quad \leq \text{reflexive closure of } < \\
\text{shape} \vdash c \neq c & \quad \leq \text{complement of } \leq \\
\text{shape} \vdash c \neq c'' \quad \text{or} \quad c = c'' & \quad \text{shape} \vdash c' \leq c'' \\
\text{shape} \vdash c'' \quad \text{extends} \ldots \text{NOTNEARERSUPER} \quad \text{c} \quad \text{extends} \quad c' \\
\text{shape} \vdash c \quad c \leq c' & \quad \text{c' extends} \ldots m[\text{c}](c_i) \quad \in \text{shape} \\
\text{shape} \vdash m \neq \text{Object} & \quad \text{c extends} \quad c' \quad \text{mdecl} \\
\text{shape} \vdash m \notin c & \quad \text{shape} \vdash c' \quad \text{Satisfies} \quad \text{c extends} \quad c' \quad \text{c', m[c](c_i)} \\
\end{align*}
\]

Figure 14: MiniJava type and method assignment judgments

In addition to the import shapes (\( \text{shape}_i \)) and export shapes (\( \text{shape}_e \)), given in the syntax of a compound or atom, three other sequences of shapes are considered in the reductions and judgments. The signature shapes of a unit (\( \text{shapes}_i \)) are the shapes of imports and exports that are visible to the enclosing compound expression. The defined shapes of a unit (\( \text{shapes}_d \)) are the shapes of a unit expression's class definitions. To get the defined shapes of an atom expression, the operator \( [\vdash] \) (definition omitted from the figure) extracts a shape from a class by dropping method overrider, and turning fresh methods into method declarations. To get the set of defined shapes in a compound expression, the shapes of exports of local unit expressions are used. The declared shapes of a unit (\( \text{shapes}_d \)) are the shapes used in type checking the class definitions inside the unit; they are merely the combination of the import and the defined shapes.

Two judgments are omitted from Figure 15. The method declaration judgment (\( \text{-\textit{decl}} \)) ensures that a method declaration is well formed: classes referred to by a method declaration's return and argument types must exist in the environment. The method implementation judgment (\( \text{-\textit{meth}} \)) ensures that method's implementation is a well-formed expression according to expression-level type judgments.

\[
\begin{align*}
\text{shape}_d \vdash c' & \quad m'[c(i)](c_i) \in c' \\
\text{shape}_d \vdash \text{c} \quad \text{meth} \quad c' \quad m'[c(i)](c_i) \in \text{expr} \\
\text{shape}_d \vdash \text{c} \quad \text{meth} \quad c' \quad m'[c(i)](c_i) \in \text{expr}' \\
\text{shape}_d \vdash \text{c} \quad \text{meth} \quad c' \quad f \quad \text{m}[c(i)](c_i) \in \text{expr} \quad \text{fresh} \\
\end{align*}
\]

Figure 15: MiniJava unit checking rules

In Figure 15, the introduction judgment (\( \text{-\textit{intro}} \)) checks both theshapes of imports with respect to the signature shapes and the shapes of defined classes with respect to the declared shapes. It ensures that the superclasses of a class is bound and that fresh method declarations are well formed. The rule also ensures that circularity does not occur, and that method collisions do not occur by checking that fresh methods have not been declared in superclasses.

The export judgment (\( \text{-\textit{export}} \)) ensures that an export has a bound class definition with the correct (but not maximal) inheritance relationship, and that its superclass is bound in the signature shapes. All required methods in the export's shape must be introduced in a class definition that is the same or superclass of the bound class definition. However, methods fresh in the shape of the export must not be implemented in the
visible superclass of the export. Additionally, the export rule ensures that method declarations are well formed.

The definition judgment \( \triangleright \text{def} \) ensures that methods overridden in a class are declared in the class’s superclass and that method implementations well formed.

As shown in Figure 17, atom expressions are checked by applying the introduction, export, and the definition judgments along with the \textsc{NotNearerSuper} rule.

\[
\begin{align*}
\vdash \text{atom} \ : \ \text{sig} \ & \text{[import } \text{shape}_e \text{]} \text{[import } \text{shape}_a \text{]} \text{[export } \text{shape}_e \text{]} \text{[export } \| \text{[shape]} \| \text{]} \text{[body } \text{class} \text{]} \\
\text{shape} = \text{shape}_a \text{[extends } \text{c}_a \text{]} \text{[extends } \text{class} \text{]} \text{[extends } \text{shape}_e \text{]} \\
\text{shape}_a = \text{shape} \text{[extends } \text{c}_a \text{]} \\
\text{shape}_e = \text{shape}_a \text{[extends } \text{c}_e \text{]} \\
\text{shape}_a = \text{shape} \text{[extends } \text{shape}_a \text{]} \\
\text{shape} = \text{shape} \text{[extends } \text{shape}_e \text{]} \\
\text{c}_a \text{ distinct } \text{c}_e \\
\text{shape} = \text{shape}_a \text{[extends } \text{shape}_e \text{]} \\
\vdash \text{compound} \ : \ \text{sig} \ & \text{[import } \text{shape} \text{]} \text{[import } \text{shape} \text{]} \text{[export } \text{shape}_e \text{]} \text{[export } \| \text{[shape]} \| \text{]} \text{[make } \text{atom}(c) \text{]} \\
\text{shape} = \text{c}_e \text{[extends } \text{...} \text{]} \text{[extends } \text{...} \text{]} \text{[extends } \text{...} \text{]} \text{[extends } \text{...} \text{]} \text{[extends } \text{shape}_e \text{]} \\
\end{align*}
\]

\[
\begin{align*}
\text{atom} = \text{atom} \ [\text{import } \text{shape}_e \text{]} \text{[export } \text{shape}_a \text{]} \text{[body } \text{class} \text{]} \\
\text{shape}_a = \text{c}_a \text{[extends } \text{...} \text{]} \\
\text{shape} = \text{c}_a \text{[extends } \text{...} \text{]} \\
\text{class} = \text{c}_a \text{[extends } \text{...} \text{]} \\
\text{shape}_a = \text{c}_a \text{[extends } \text{...} \text{]} \\
\text{shape} = \text{c}_a \text{[extends } \text{...} \text{]} \\
\end{align*}
\]

Figure 16: Compound reduction

Figure 17: Atom type checking

Figure 18: Compound type checking

As shown in Figure 18, Compound expressions are checked by ensuring that classes linked to the imports of local unit expressions are either exports of other local unit expressions or imports of the compound expression. An updated set of shapes for exports and imports for all local unit expressions is derived by replacing references to imported classes with references to the linking classes. These updated exported shapes are then used as the compound’s defined shapes. The declared shapes are used to ensure that classes linked to imports of local unit expressions \textsc{Satisfy} the shape of the import. Additionally all checks that are applied to atom expressions, except for the definition judgment, are also applied to compound expressions on the corresponding shapes.

Typing an atom or compound expression provides a signature. When computing signatures, the \([\bullet]\) operator transforms the exported shapes of a unit by hiding the actual source of a exported method: it changes \( c \text{ extends } c' \{ \ldots m[c] \ldots \} \) to \( c \text{ extends } c' \{ \ldots m[c] \ldots \} \), since the name \( c \) is necessarily private to the unit if it is not \( c \) itself. Only these signatures are used when checking the composition of local unit expressions in enclosing compound expressions.

In MiniJiazzi, the actual classes linked to imports can have relationships with other classes that are not implied by the shape of the import. This is in contrast to Jiazzi, which must disallow undetected subclassing relationships to prevent problems with overloading in the Java language and allow local reasoning about runtime coercions.

3.2. Reduction (Linking) Rules

The \textsc{compound} reduction rule \((\rightarrow_c)\), shown in Figure 16, performs unit linking by reducing a compound unit expression to an atomic unit expression. This reduction applies only after each of the local unit expressions has been reduced to an atomic unit expression.

Whereas the type rule requires that exports have unique names, the \textsc{compound} reduction requires that all classes defined by its local units have unique names. Thus, before applying the compound reduction, it may be necessary to rename internal classes. The rule for renaming methods, omitted from the figure, is that any class absent from the interface of an atom expression can be renamed.

To reduce a compound, the imports of the compound unit expression are copied intact to the new atomic unit
expression. Each local unit expression is replaced by the sequence of class definitions contained in the local unit expression after they are renamed. These definitions must be renamed in two ways. First, the names of formal imports are replaced by the names of the actual imports. Second, the source class for imported methods is updated to reflect the more precise source information made available during linking. The export clause is similarly updated to reflect more precise method-source information.

3.3. Soundness
Since the type-checking rule for unit expressions in MiniJiazzi relies only on local import and export declarations, and since the compound checking relies only on the signatures of its local units, the MiniJiazzi model shows how Jiazzi supports separately checked components that import and export classes. The proof of the soundness theorem for MiniJiazzi (Figure 11) is available in our technical report, and the soundness of MiniJiazzi sketches a soundness claim and proof for the full Jiazzi language.

4. Jiazzi Implementation
One of our goals in building Jiazzi was to avoid modifying the core Java language. Therefore, Jiazzi is a separate component description and linking language. The Jiazzi implementation takes unit definitions written in the Jiazzi language and Java class files, and generates new Java class files that are compatible with standard Java Virtual Machines. We have an implementation of Jiazzi that can build all of the examples in Section 2.

Classes in atoms can come from standard Java class files or source files. If classes come from source files, the Jiazzi implementation compiles these files against stub classes that represent the imports of the atom. The stub classes are automatically generated from the imports’ shapes. The stubs are only used for supporting source compilation with standard Java compilers, and can be discarded after source compilation.

In Java, classes declare imports by embedding the names of imports inside their class file. In Jiazzi, these class imports can correspond to the containing unit’s imports or peers (classes implemented in the same unit). However, for purposes of backwards compatibility, Jiazzi allows some class imports to be unbound from the perspective of the unit system. These unbound classes are obtained using Java’s own namespace management mechanism. As the expressiveness of Jiazzi matures, we expect that all class imports will be able to correspond to either imports or peer classes.

When a unit is instantiated inside a compound, its bytecode is duplicated. References to imports in the duplicated bytecode are replaced according to how the imports are linked in the enclosing compound. If a class or method is not visible outside of the compound, it is renamed so that its name will not collide with other names that are added in different scopes. In our implementation, this is done by adding the name of the enclosing local unit to the name of the hidden class or method each time it is duplicated. Classes that are exported outside of the compound are renamed according to the name of the export so that they can be found by other compounds that later link the current compound. Methods visible outside of the compound retain their visible name. The bytecode derived in all local units of a compound becomes the bytecode for the compound when it is itself linked by another compound, or loaded for execution.

```java
class locked$key$KeyDoor
extends Door {
    Object neededPass$locked() {...}
    void setKey(Key k) {...}
}

class locked$secure$SecureDoor
extends locked$key$KeyDoor {
    boolean canOpen(Person p) {
        o = this.neededPass$locked();
        if (!p.hasItem(o)) return false;
        else return super.canOpen(p);
    }
}

class magic$spell$SpellDoor
extends locked$secure$SecureDoor {
    Object neededPass$magic() {...}
    void setSpell(Spell s) {...}
}

class LockedMagicDoor
extends magic$spell$SpellDoor {
    boolean canOpen(Person p) {
        o = this.neededPass$magic();
        if (!p.hasItem(o)) return false;
        else return super.canOpen(p);
    }
}
```

Figure 19: This is the decompiled source code for the class LockedMagicDoor as linked in Figure 10.

Figure 19 shows how renaming works for the hidden method names in Figure 9 in the creation of the LockedMagicDoor class. Since the neededPass methods are hidden in both the locked and magic local units respectively, they are qualified with the names of those local units respectively when the bytecode for the compound definition of LOCKED-MAGIC-CORRECT is generated.

Duplicating and rewriting bytecode directly leads to a simple implementation of Jiazzi. We avoid the overhead of establishing virtual lookup tables and
object layouts that can be used within any linking context. Each duplicated version of the unit can be specialized separately by an optimizing compiler according to how its imports are connected. However, duplication results in an increase in the code footprint and increases the time spent in post-bytecode compilation. Future work will focus on reducing code duplication while still maintaining efficiency.

Our implementation of Jiazzi supports Java interfaces. For simplicity, we do not allow methods in interfaces to be hidden across component boundaries since they are abstract. Additionally, methods can only be introduced into the shape of a class once. Some valid Java constructions can violate this rule. For example, assume unrelated interfaces $I_0$ and $I_1$ each contain method $m$ with the same signature. In Java, it is valid for a class $C$ to implement both $I_0$ and $I_1$. Since methods are renamed at their introduction point, these constructions cannot be represented in Jiazzi; otherwise, after renaming it would be possible for a method implementation to override multiple methods with different names. As a result, these constructions cannot be described in a unit’s signature, though they can still be used inside the unit’s implementation.

5. Related Work

Moby [15][16] is a structurally typed object oriented language that supports ML-style modules. Methods can be hidden from modules as in Jiazzi. Object types created in these modules will not propagate the hidden methods in their type. However, the hidden methods can still be invoked by explicitly specifying its originating class types. Since Moby does not use subclasing relationships implicitly when typing method invocations, module applications that create method collisions are allowed. To resolve ambiguous methods, Moby relies on object-view coercion to explicitly coerce the type of an expression from a class to one of its superclasses.

Moby’s module system does not support mutually dependent modules, an important feature of Jiazzi. If Moby were extended with mutually recursive modules, we expect that Jiazzi’s maximal subtyping rules would be necessary for module signatures [11].

The module system in Objective Caml [23], like Jiazzi, supports external connections of classes. Since classes can be defined in modules, these classes can also form something like mixins. However, Objective Caml does not permit a class supplied to a component (functor) to provide more methods than required by the component.

JavaMod [4], like Jiazzi, supports the import and export of classes and mutually dependent modules without the danger of cyclic inheritance. Support of method hiding in imported classes is not supported, so a JavaMod module cannot create useful mixin constructions. Additionally, the subtyping relationships of imports cannot be expressed in a JavaMod module.

Modula-3 [10] supports modules that can import and export methods. However, modules can be linked only once per program. ComponentJ [27] is a unit-like component system for Java that supports the import and export of methods across component boundaries. The surface syntax of ComponentJ influenced revisions of Jiazzi’s surface syntax.

Mixins were pioneered in the Common Lisp Object System [21] and further modeled by Bracha [5], Ancona and Zucca [2], and Flatt et al. [18]. Mixins have also been implemented in Smalltalk [6] and Beta [13]. JAM [1] adds mixins to the Java language as an extension. It supports common types for mixins, which Jiazzi does not. However, JAM does not consider mixins with other parameterized types other than the superclass. Because of class aliasing concerns, JAM must restrict the use of this within the mixin. There is no such restriction in Jiazzi, since each application of a unit creates a new set of unique classes. Methods cannot be scoped in JAM, requiring an analysis of the class hierarchy when a mixin is applied to check for method collisions.

Component standards such as COM [24], JavaBeans [30], and CORBA [26] define components as objects, so they do not support the usage of important language mechanisms (such as inheritance) across components.

6. Conclusion

We have presented the design and semantics of Jiazzi, a powerful and practical component language for unmodified Java. Class derivations can span component boundaries. Despite the fact that each individual component can be implemented, type-checked, and compiled before the source of its imports has been determined, Jiazzi ensures that component compositions produce class hierarchies without inheritance cycles or colliding method definitions.

Jiazzi is still evolving. We are currently adding direct support for class frameworks based on the reverse mixin pattern. In addition, we are enhancing the language to allow shapes to be reused in many units, which will make the system more practical for production use. We are also adding support for the rest of the Java language (e.g., static methods and fields) so that more constructs can be exposed in a unit’s signature.
7. Bibliography


