Jiazzi: New-Age Components for Old-Fashioned Java

Sean McDirmid, Matthew Flatt, Wilson C.Hsieh

School of Computing, University of Utah

Presented by Xiaoxia Ren

Component-based Development

- Programming with “modules” whose external references are decoupled from particular implementations.
  - Code reuse
- Jiazzi – independently compiled “units” can be linked to form a complete application

Java packages

- The classes in each package are coupled to the imported packages.

Java packages

- The classes in each packaged are coupled to the imported packages.

```
package ui
public class Widget{
 public void paint(){…}
 …
}
public class Button extends Widget{
 public void setLabel(String s){…}
 …
}
public class Window extends Widget{
 public void add(Widget w){…}
 public void show(){…}
 …
}
```

```
package applet;
import ui.*;
public class Program extends
Window{
  Button b = new Button();
  public Program(){
    b.setLabel(“start”);
    add(b);
  }
  public void run(){
    show();
  }
}
```

Java packages

- The classes in each packaged are coupled to the imported packages.

```
package ui
public class Widget{
 public void paint(){…}
 …
}
public class Button extends Widget{
 public void setLabel(String s){…}
 …
}
public class Window extends Widget{
 public void add(Widget w){…}
 public void show(){…}
}
```

```
package applet;
import ui.*;
public class Program extends Window{
  Button b = new Button();
  public Program(){
    b.setLabel(“start”);
    add(b);
  }
  public void run(){
    show();
  }
}
```

- class Program can’t compile separately
- It’s difficult to replace package references to other package references that provide the same functionality

Jiazzi

- Support separate compilation
  - Enables development of large programs and deployment of components in binary form
- Support external linking
  - Eliminates hard-coded dependencies to make components flexible
- units: import packages → export packages

Contributions

- Jiazzi integrates with Java using a "stub generator" and an "external linker".
  - No special core language extensions needed to construct a component.
- Support the addition of features to classes without editing their source code or breaking existing class variant relationships.

Package Signatures (1)

```
package ui
public class Widget{
 public void paint(){…}
 public void draw(){…}
}
public class Button extends Widget{
 public void setLabel(String s){…}
  .. .. ..
}
public class Window extends Widget{
 public void add(Widget w){…}
 public void show(){…}
  .. .. ..
}
```

File: ./ui_s.sig

```
signature ui_s{
 class Widget
 { void paint(); }
 class Button extends Widget
 { void setLabel(String s); }
 class Window extends Widget
 { void add(Widget w);
   void show();
 }
}
```

Package Signatures (2)

```
package ui
public class Widget{
 public void paint(){…}
  .. .. ..
}
public class Button extends Widget{
 public void setLabel(String s){…}
  .. .. ..
}
public class Window extends Widget{
 public void add(Widget w){…}
 public void show(){…}
  .. .. ..
}
```

File: ./ui_s.sig

```
signature ui_s{
 class Widget
 { void paint(); }
 class Button extends Widget
 { void setLabel(String s); }
 class Window extends Widget
 { void add(Widget w);
   void show();
 }
}
```

```
package applet;
import ui.*;
public class Program extends Window{
  .. .. ..
 public void run()
  { show(); }
}
```

File: ./applet_s.sig

```
signature applet_s<ui_p>{
 class Program extends ui_p.Window
 { void run(); }
}
```

Package Signatures (3)

```
package ui
public class Widget{
 public void paint(){…}
  .. .. ..
}
public class Button extends Widget{
 public void setLabel(String s){…}
  .. .. ..
}
public class Window extends Widget{
 public void add(Widget w){…}
 public void show(){…}
  .. .. ..
}
```

File: ./ui_s.sig

```
signature ui_s<st_p>{
 class Widget extends Object
 { void paint(); }
 class Button extends st_p.Widget
 { void setLabel(String s); }
 class Window extends st_p.Widget
 { void add(st_p.Widget w);
   void show();
 }
}
```

```
package applet;
import ui.*;
public class Program extends Window{
  .. .. ..
 public void run()
  { show(); }
}
```

File: ./applet_s.sig

```
signature applet_s<ui_p>{
 class Program extends ui_p.Window
 { void run(); }
}
```

Atoms

File: ./ui_s.sig

```
signature ui_s<st_p>{
 class Widget extends Object
 { void paint(); }
 class Button extends st_p.Widget
 { void setLabel(String s); }
 class Window extends st_p.Widget
 { void add(st_p.Widget w);
   void show();
 }
}
```

File: ./applet_s.sig

```
signature applet_s<ui_p>{
 class Program extends ui_p.Window
 { void run(); }
}
```

File: ./applet.unit

```
atom applet {
 import ui_in: ui_s<ui_in>;
 export applet_out: applet_s<ui_in>;
}
```

Java Source of “applet_out.Program”

File: ./applet/applet_out/Program.java

```
package applet_out;
public class Program extends
ui_in.Window{
  ui_in.Button b =
        new ui_in.Button();
  public Program(){
    b.setLabel(“start”);
    add(b);
  }
  public void run(){
    show();
  }
}
```

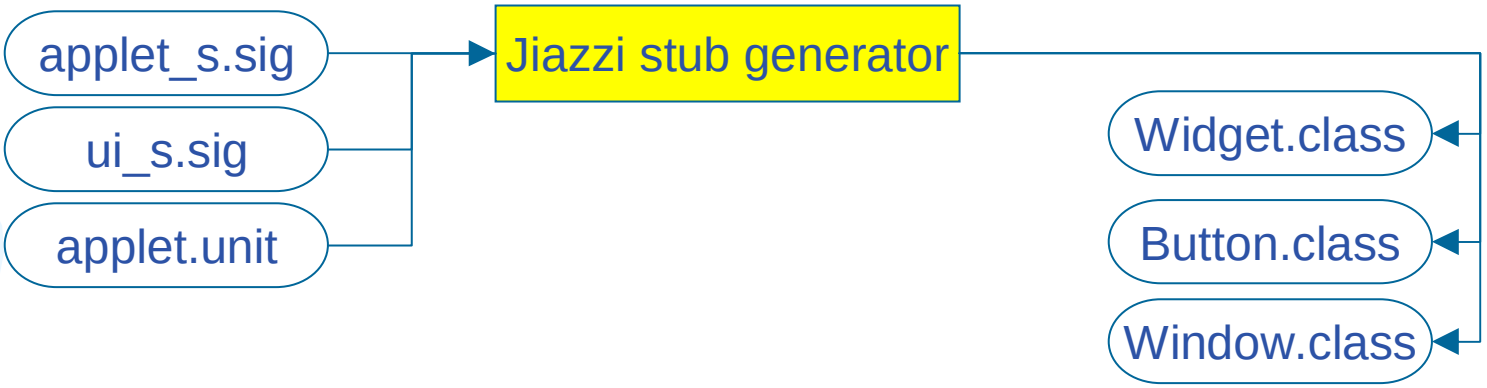
Java Source of “applet_out.Program”

File: ./applet/applet_out/Program.java

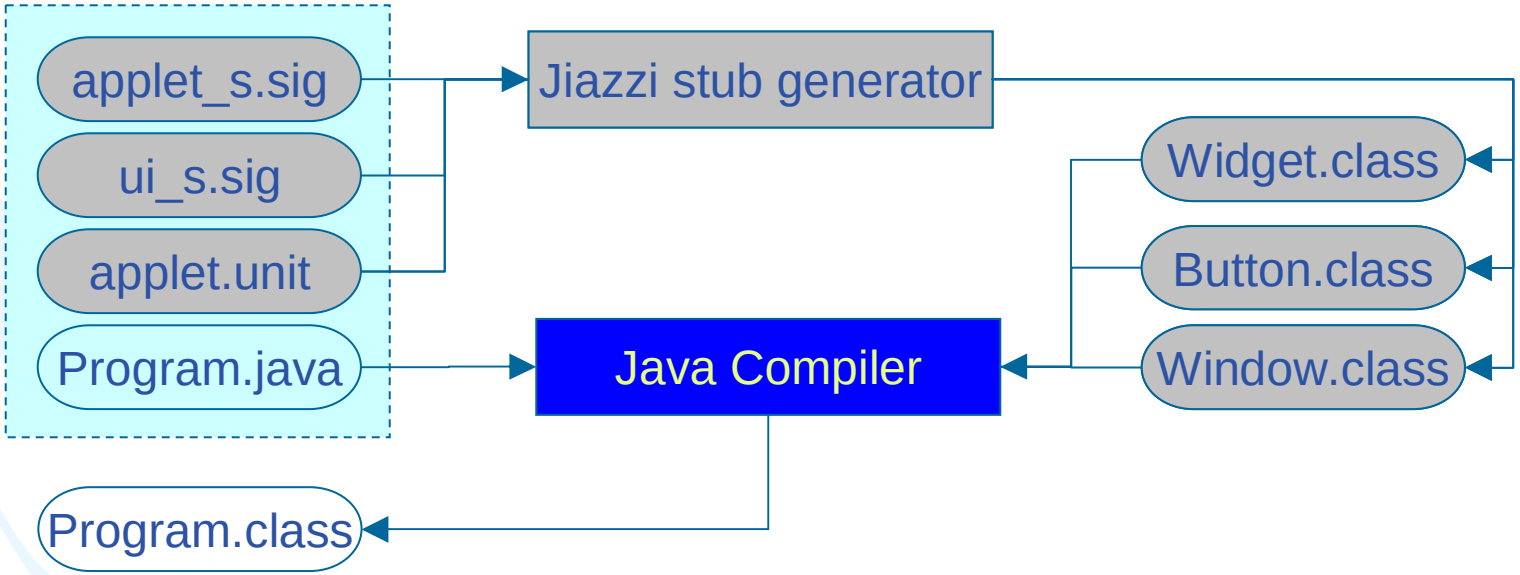
```
package applet_out;
public class Program extends
ui_in.Window{
  ui_in.Button b =
        new ui_in.Button();
  public Program(){
    b.setLabel(“start”);
    add(b);
  }
  public void run(){
    show();
  }
}
```

```
package applet;
import ui.*;
public class Program extends
Window{
  Button b = new Button();
  public Program(){
    b.setLabel(“start”);
    add(b);
  }
  public void run(){
    show();
  }
}
```

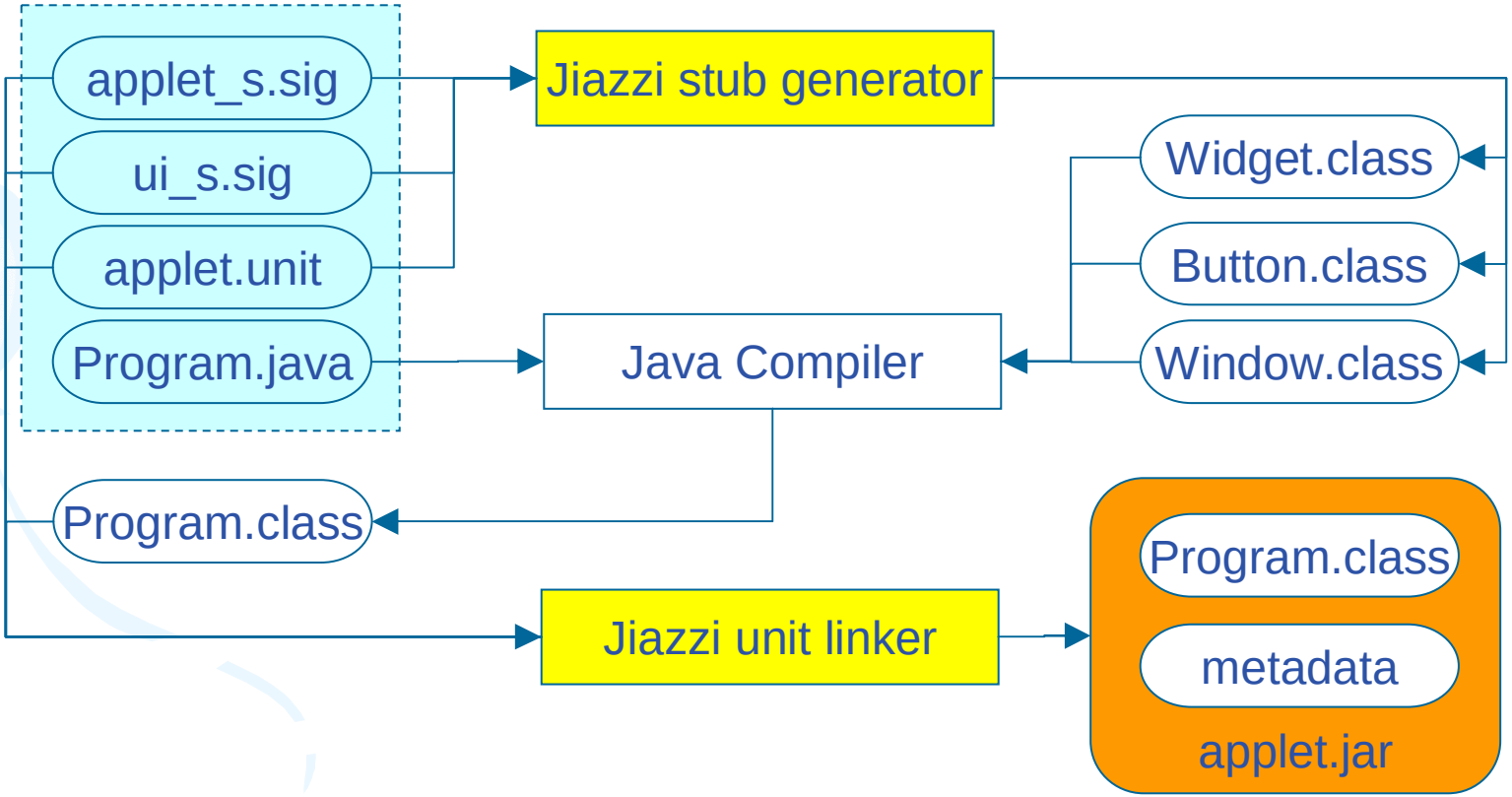
Development of “applet”



Development of “applet”



Development of “applet”



Compounds

File: ./ui.unit

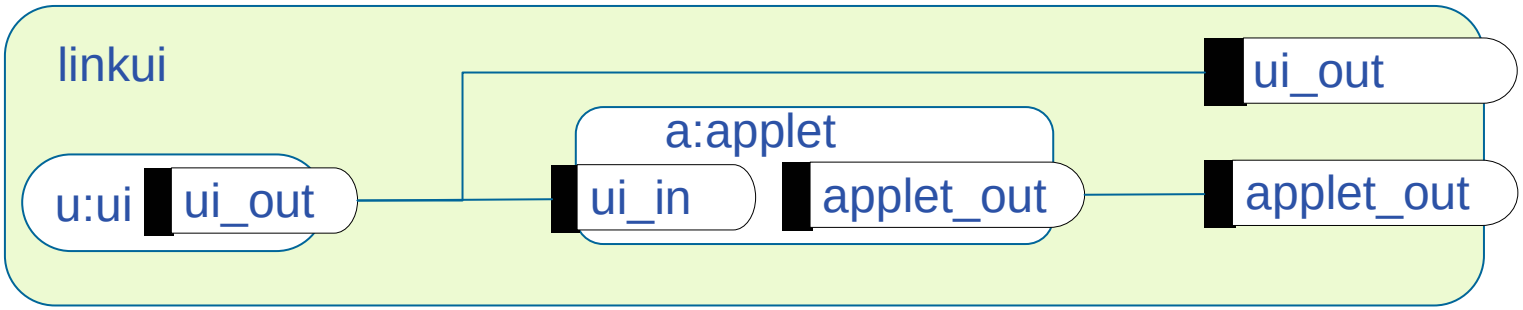
```
atom ui {
 export ui_out: ui_s<ui_out>;
}
```

File: ./applet.unit

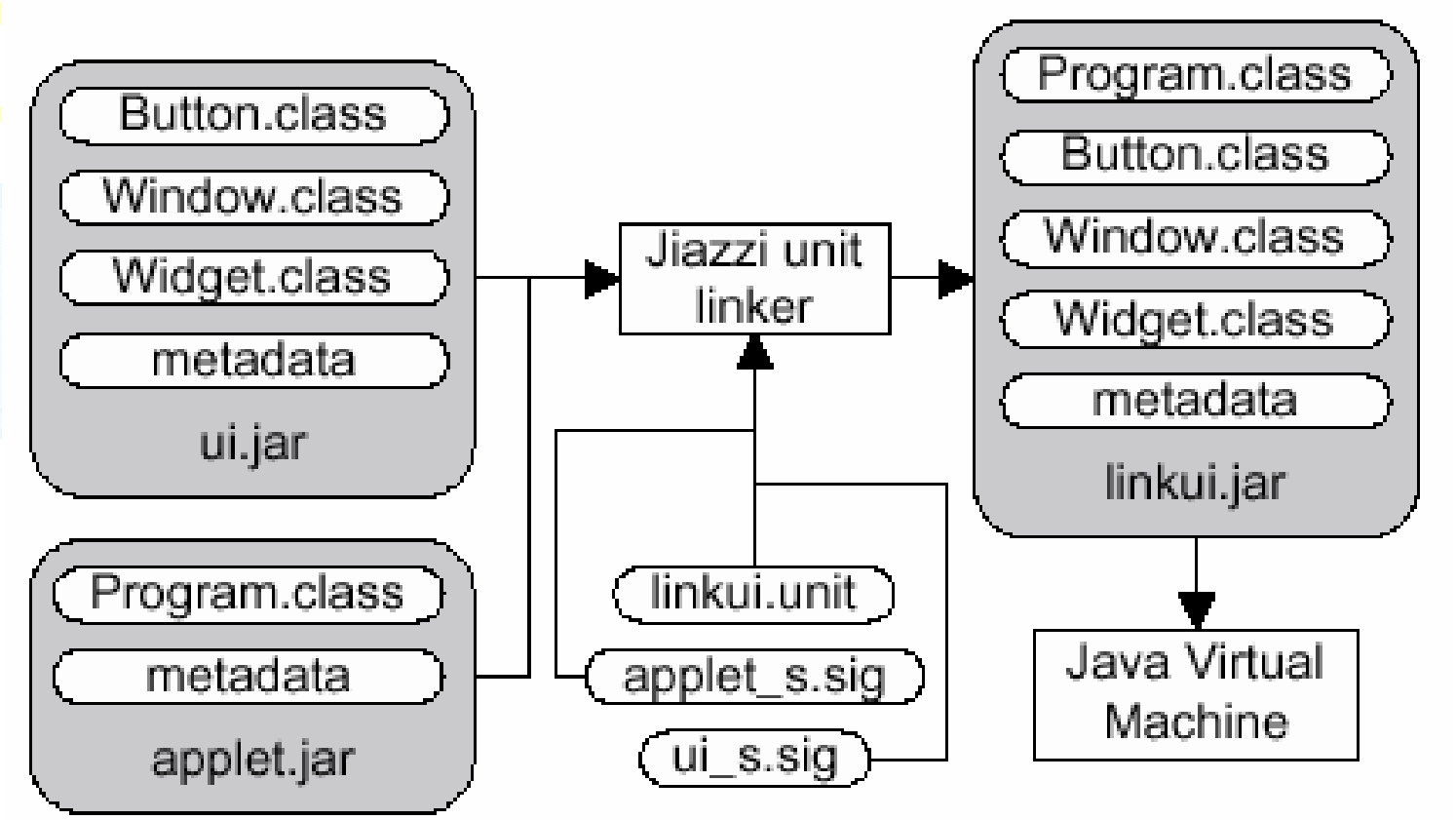
```
atom applet {
 import ui_in: ui_s<ui_in>;
 export
  applet_out: applet_s<ui_in>;
}
```

File: ./linkui.unit

```
compound linkui {
 export ui_out: ui_s<ui_out>,
        applet_out: applet_s<ui_out>;
}{
 local u:ui, a:applet;
 link u@ui_out to a@ui_in,
      u@ui_out to ui_out,
      a@applet_out to applet_out;
```



Development of “linkui”



Expressiveness

- Mixin approach (utilizes mixins to add features to classes)
- Open class pattern (use cyclic linking to solve the extensibility problem)

History of Mixins

- Came from Steve's Ice Cream at MIT.
- In PL, *mixin* first used in Lisp community,
  - a design pattern to try to control the trouble of the **multiple inheritance** of that language
- Java sticks to single inheritance
  - copied code along multiple branches of the inheritance hierarchy
- Mixin
  - Abstract subclass
  - Or: classes → new subclasses

Mixin Example in JAM

```
class Pane {
  ...
  void setVisible(boolean value) { ... }
}
```

Mixin Example in JAM

```
class Pane {
  ...
  void setVisible(boolean value) { ... }
}
```

```
class DialogBox {
  ...
  void setVisible(boolean value) { ... }
}
```

Mixin Example in JAM

```
class Pane {
  ...
  void setVisible(boolean value) { ... }
}
```

```
class DialogBox {
  ...
  void setVisible(boolean value) { ... }
}
```

```
mixin Scrollable {
  int maxScrollSize;
  inherited void setVisible(boolean value);
  void doScroll(){…}
}
```

Mixin Example (Continue)

```
class Pane {
  ...
  void setVisible(boolean value) { ... }
}
```

```
class DialogBox {
  ...
  void setVisible(boolean value) { ... }
}
```

```
mixin Scrollable {
  int maxScrollSize;
  inherited void setVisible(boolean value);
  void doScroll(){…}
}
```

```
class ScrollPane =
      Scrollable extends Pane{
  ScrollPane(int maxScrollSize){
    this.maxScrollSize = maxScrollSize;
  }
}
```

```
class ScrollDialog =
      Scrollable extends DialogBox {
  ScrollDialog() {
    this.maxScrollSize = 10;
  }
}
```

Mixins in Jiazzi

- Enable reuse of class implementation
  - An exported class subclasses an imported class
- Do NOT provide a common type "mixin" to describe the functionality they add.
  - They are link-time abstractions that enable transparent class inheritance across component boundaries

Mixins example in Jiazzi

File: ./ui_s.sig

```
Signature ui_s<ui_p>{
 class Widget extends Object
 { void paint(); }
 class Button extends ui_p.Widget
 { void setLabel(String s); }
 class Window extends ui_p.Widget
 { void add(Widget w);
   void show();
 }
}
```

File: ./nop_s.sig

```
Signature nop_s<orig_p>{
 class Widget extends orig_p.Widget{}
 class Button extends orig_p.Widget{}
 class Window extends orig_p.Widget{}
}
```

File: ./color_s.sig

```
Signature color_s<orig_p>{
 class Widget extends orig_p.Widget{
   void setColor(int);}
 class Button extends orig_p.Widget{}
 class Window extends orig_p.Widget{}
}
```

File: ./mix.color.unit

```
atom mix.color {
 import ui_init: ui_s<ui_init>,
        ui_in: nop_s<ui_init>;
 export ui_out: color_s<ui_in>;
}
```

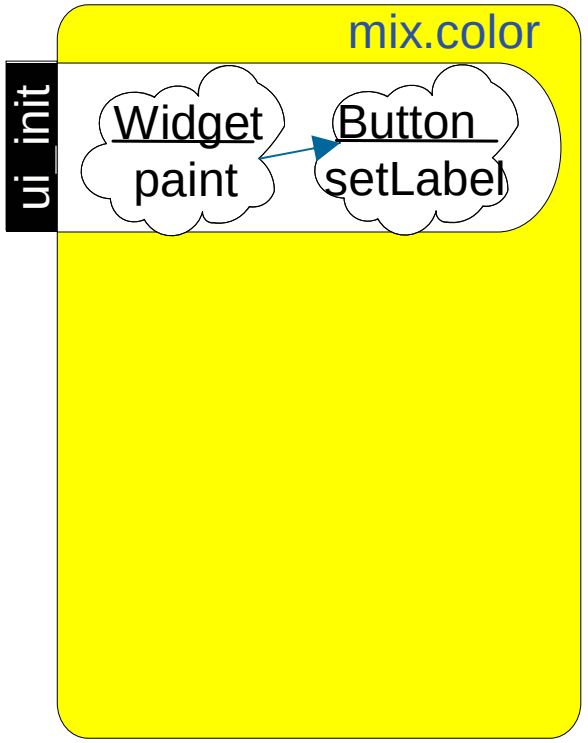
Mixins example in Jiazzi

File: ./ui_s.sig

```
Signature ui_s<ui_p>{
 class Widget extends Object
 { void paint(); }
 class Button extends ui_p.Widget
 { void setLabel(String s); }
 class Window extends ui_p.Widget
 { void add(Widget w);
   void show();
 }
}
```

File: ./mix.color.unit

```
atom mix.color {
 import ui_init: ui_s<ui_init>,
        ui_in: nop_s<ui_init>;
 export ui_out: color_s<ui_in>;
}
```



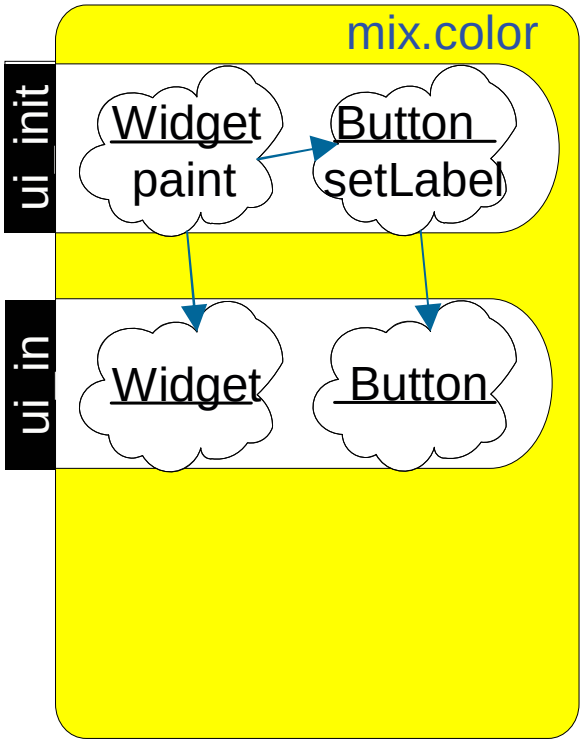
Mixins example in Jiazzi

File: ./nop_s.sig

```
Signature nop_s<orig_p>{
 class Widget extends orig_p.Widget{}
 class Button extends orig_p.Widget{}
 class Window extends orig_p.Widget{}
}
```

File: ./mix.color.unit

```
atom mix.color {
 import ui_init: ui_s<ui_init>,
        ui_in: nop_s<ui_init>;
 export ui_out: color_s<ui_in>;
}
```



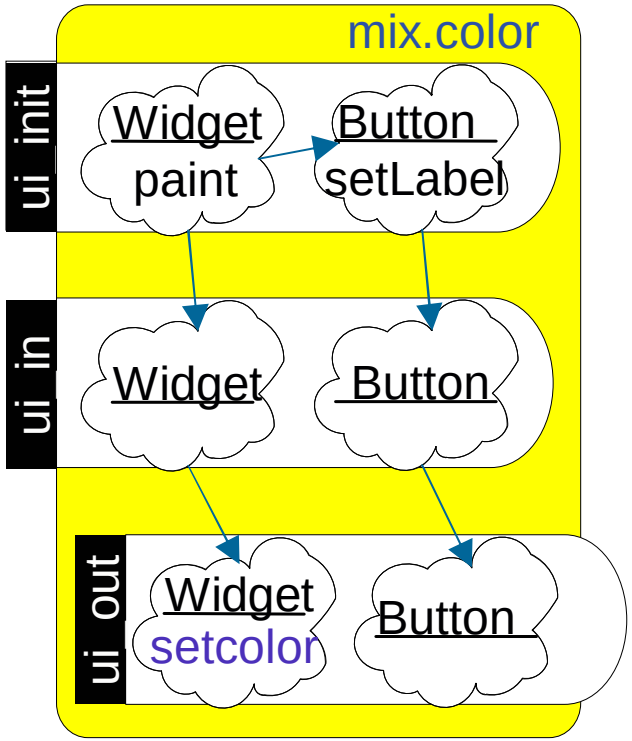
Mixins example in Jiazzi

File: ./color_s.sig

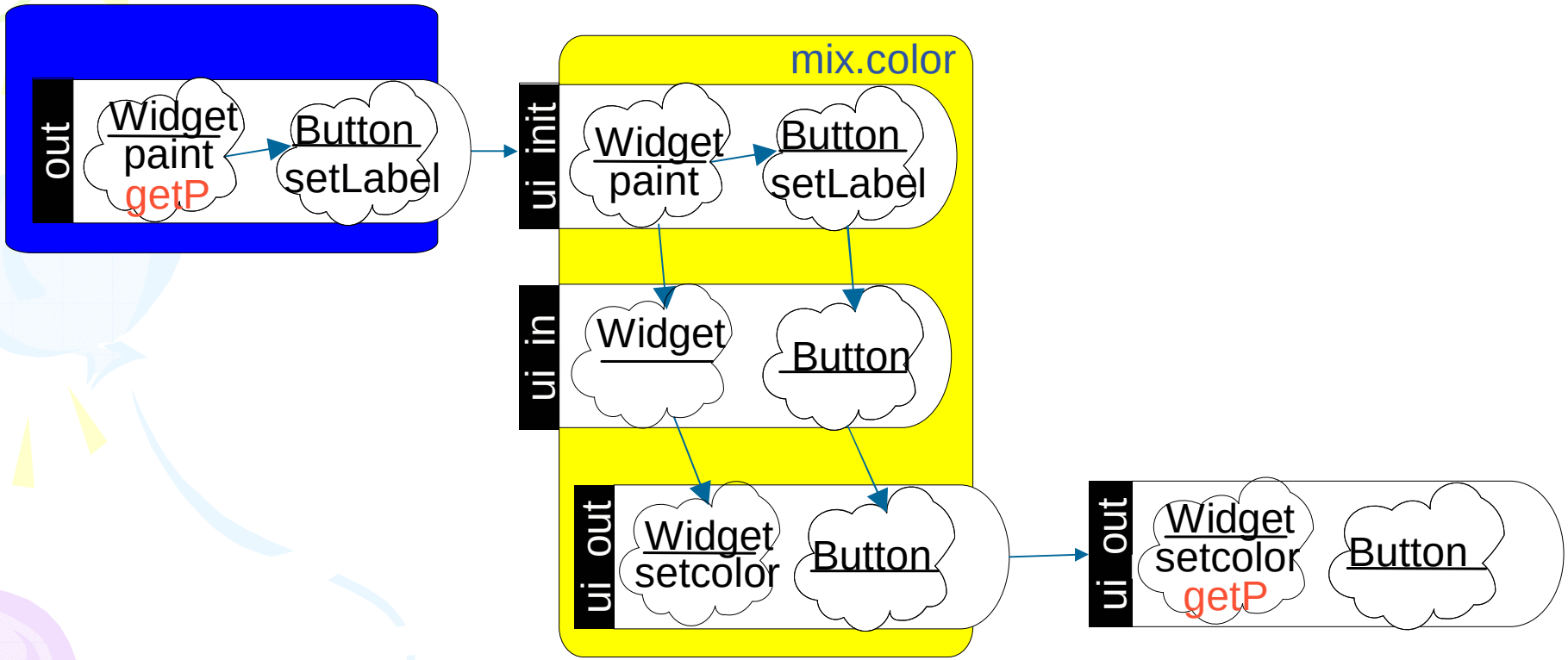
```
Signature color_s<orig_p>{
 class Widget extends orig_p.Widget{
   void setColor(int);}
 class Button extends orig_p.Widget{}
 class Window extends orig_p.Widget{}
}
```

File: ./mix.color.unit

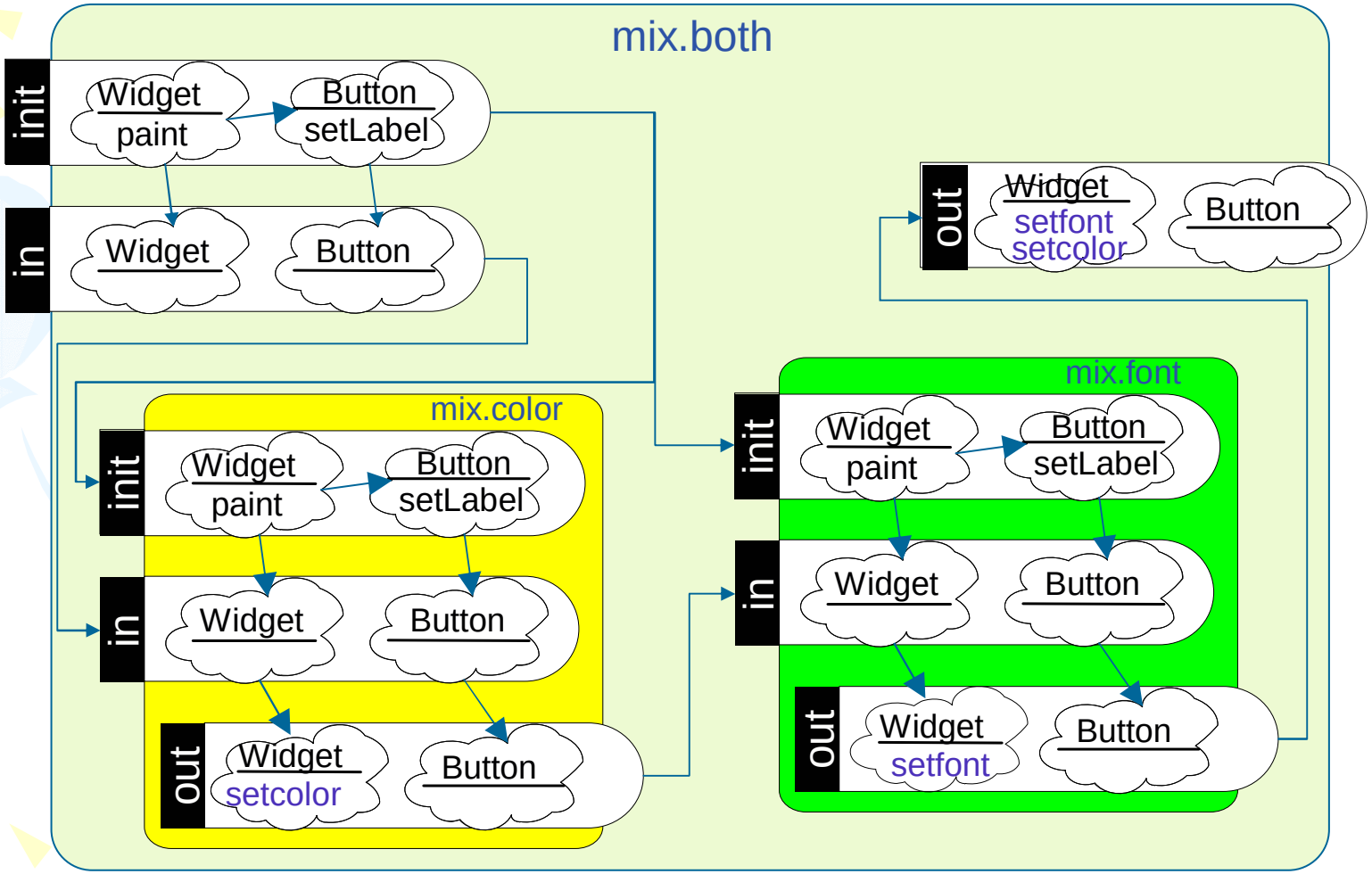
```
atom mix.color {
 import ui_init: ui_s<ui_init>,
        ui_in: nop_s<ui_init>;
 export ui_out: color_s<ui_in>;
}
```



Visibility


out
Widget
paint
getP
Button
setLabel
mix.color
ui init
Widget
paint
Button
setLabel
ui in
Widget
Button
ui out
Widget
setcolor
Button
ui out
Widget
setcolor
getP
Button

Subclassing relationships



Why ui_in?

File: ./ui_s.sig

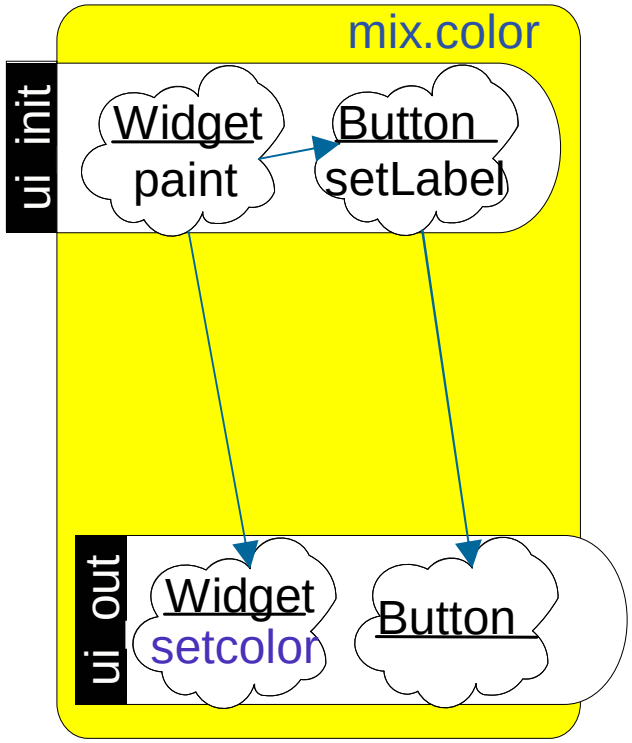
```
Signature ui_s<ui_p>{
 class Widget extends Object
 { void paint(); }
 class Button extends ui_p.Widget
 { void setLabel(String s); }
 class Window extends ui_p.Widget
 { void add(Widget w);
   void show();
 }
}
```

File: ./color_s.sig

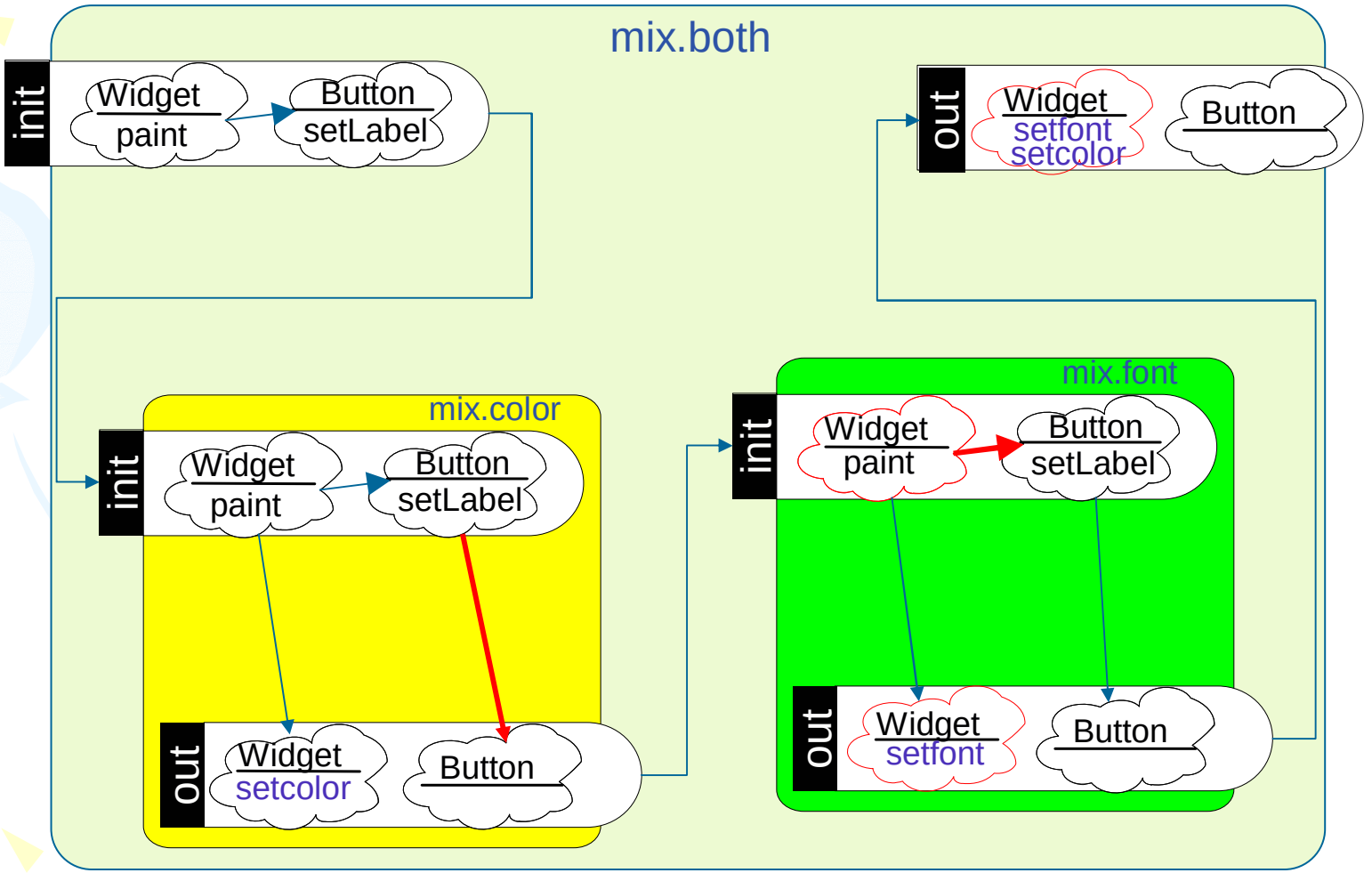
```
Signature color_s<orig_p>{
 class Widget extends orig_p.Widget{
   void setColor(int);}
 class Button extends orig_p.Widget{}
 class Window extends orig_p.Widget{}
}
```

File: ./mix.color.unit

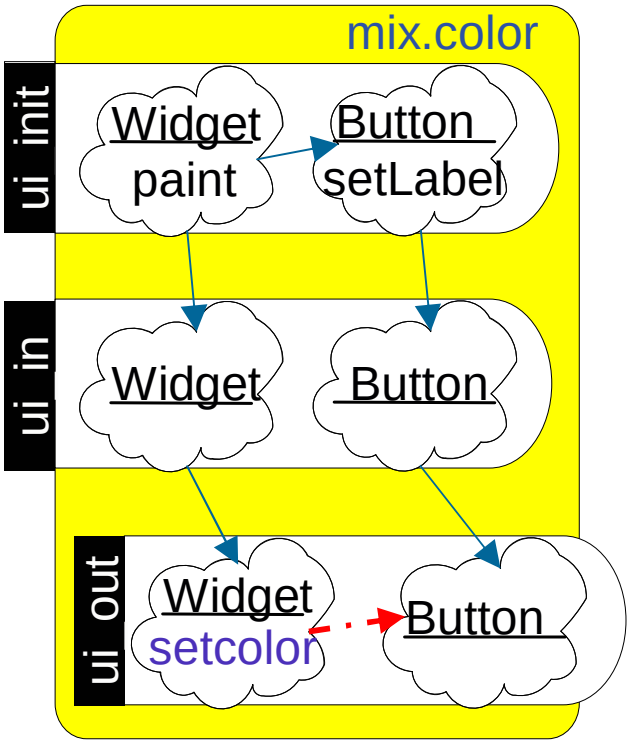
```
atom mix.color {
 import ui_init: ui_s<ui_init>,
 export ui_out: color_s<ui_init>;
}
```



No ui_in


mix.both
init
Widget
paint
Button
setLabel
out
Widget
setfont
setcolor
Button
mix.color
init
Widget
paint
Button
setLabel
out
Widget
setcolor
Button
mix.font
init
Widget
paint
Button
setLabel
out
Widget
setfont
Button

Extensibility Problem



Adding features – vertical class extension
Creating variants – horizontal class extension

} Extensibility Problem

Solution

- Not only allow the modular addition of new features to existing classes; but also ensure that added features are visible in all variants of the updated class.
  - Open class in MultiJava: allow programmers to add new methods to existing classes without creating distinct subclasses or editing existing code.
  - Jiazzi: support open class pattern to simulate open class

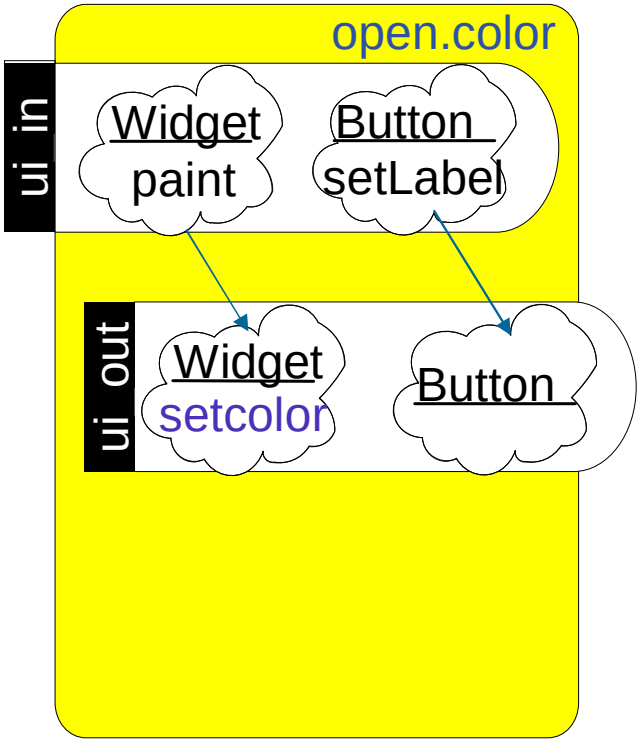
Upside-down mixins

- Imported classes subclass exported ones

A Solution

File: ./open.color.unit

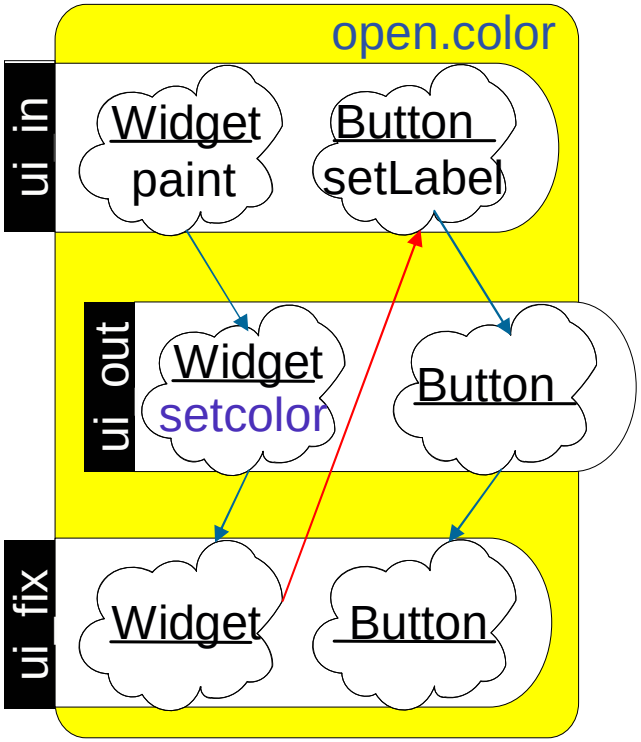
```
atom open.color {
 import ui_in: ui_s<ui_fixed>,
        ui_fixed: nop_s<ui_out>;
 export ui_out: color_s<ui_in>;
}
```



A Solution

File: ./open.color.unit

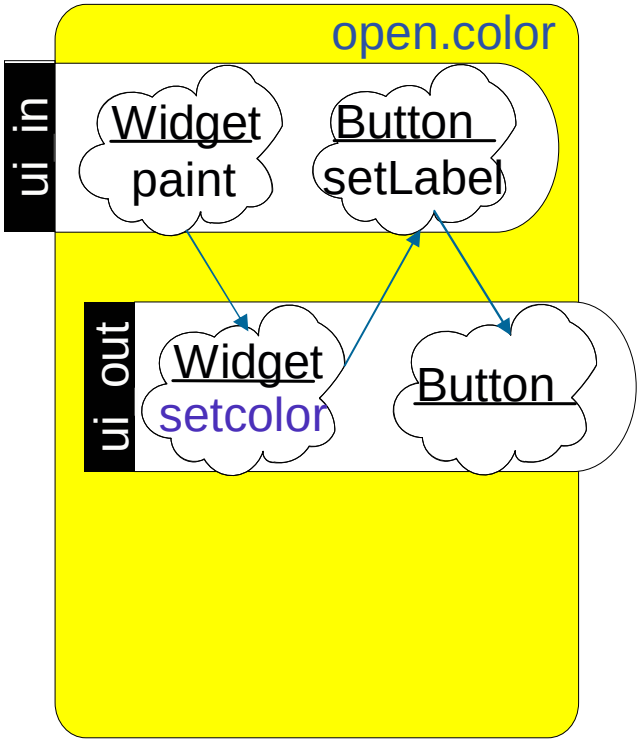
```
atom open.color {
 import ui_in: ui_s<ui_fixed>,
        ui_fixed: nop_s<ui_out>;
 export ui_out: color_s<ui_in>;
}
```



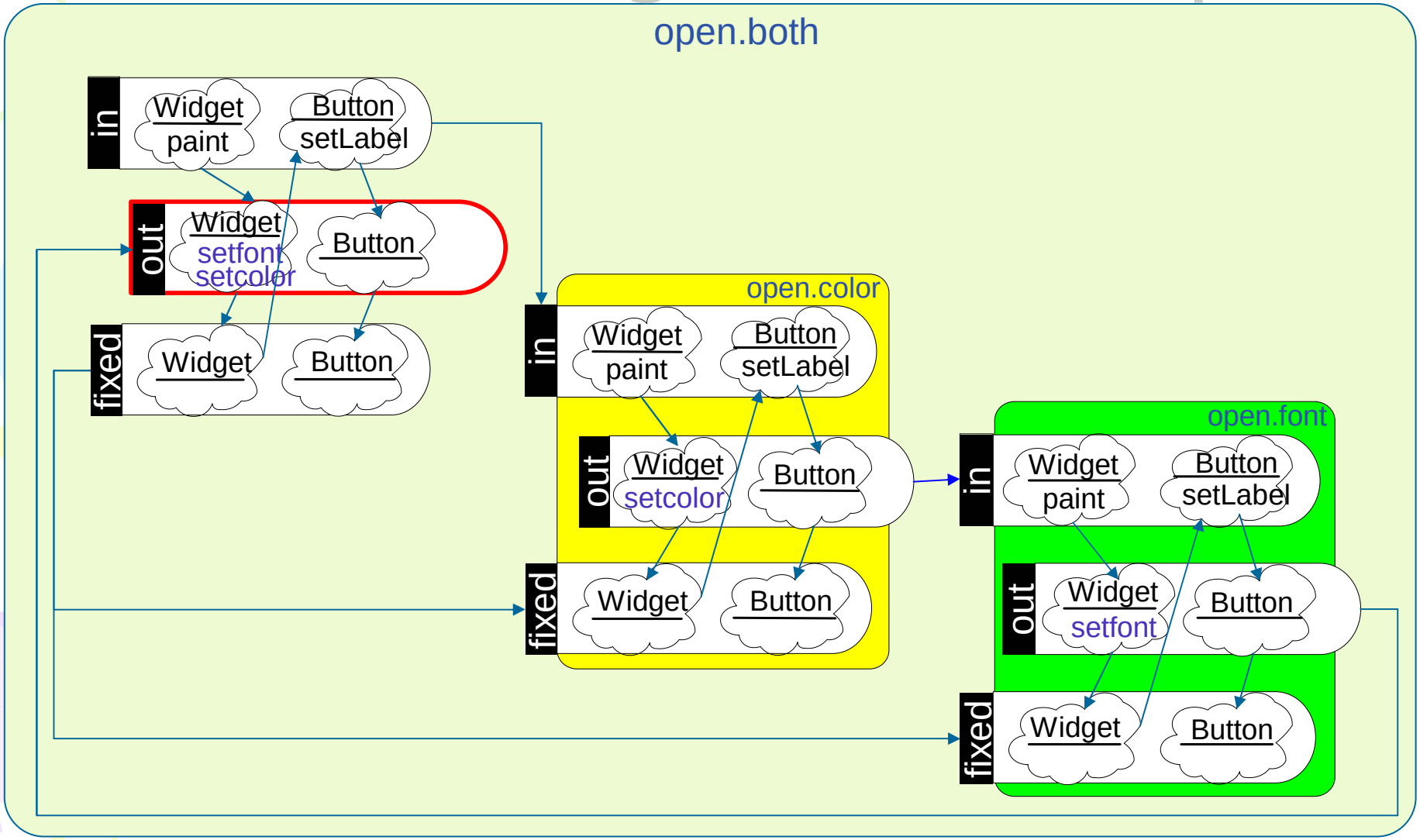
Why ui_fixed?

File: ./open.color.unit

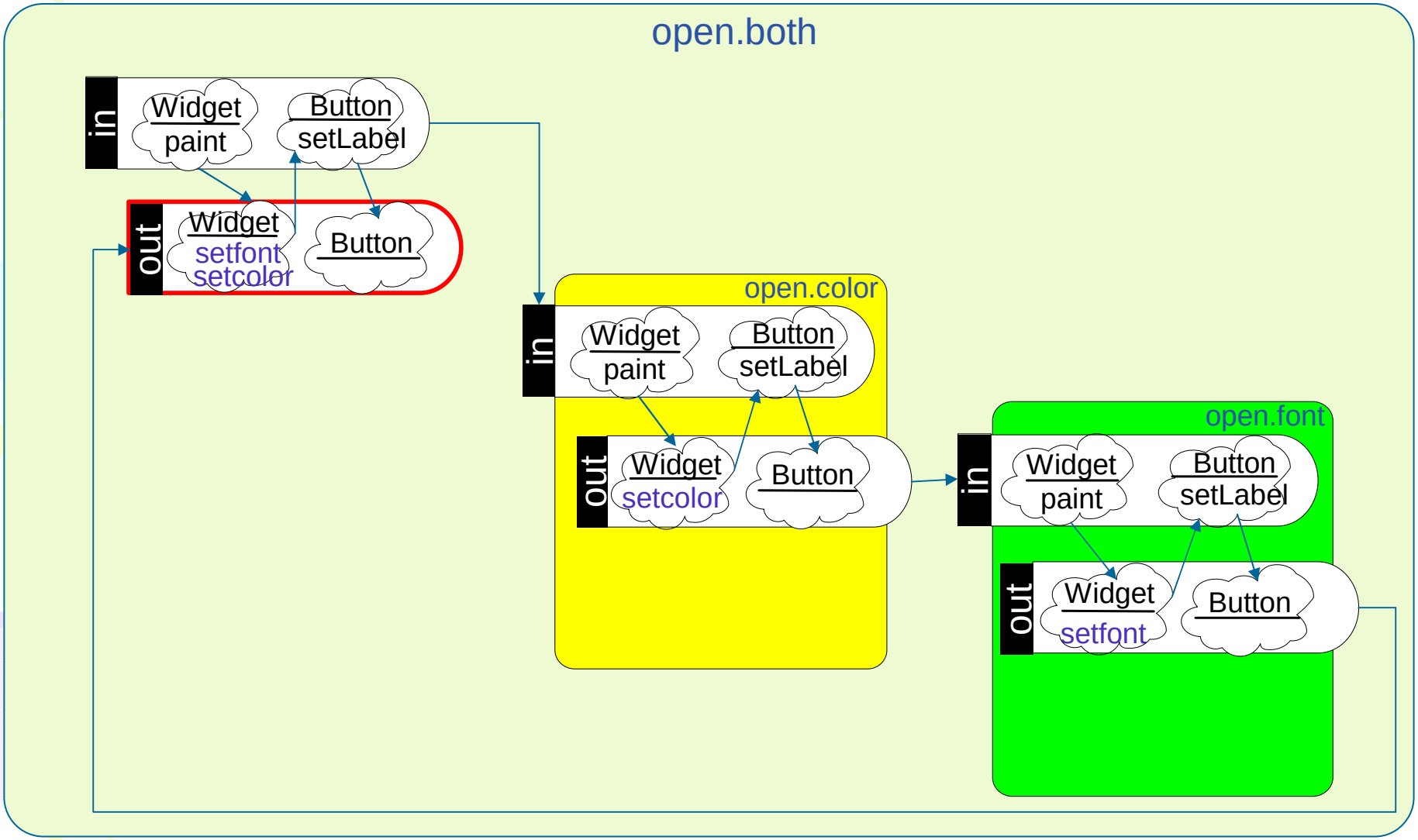
```
atom open.color {
 import ui_in: ui_s<ui_out>,
 export ui_out: color_s<ui_in>;
}
```



Subclassing relationships



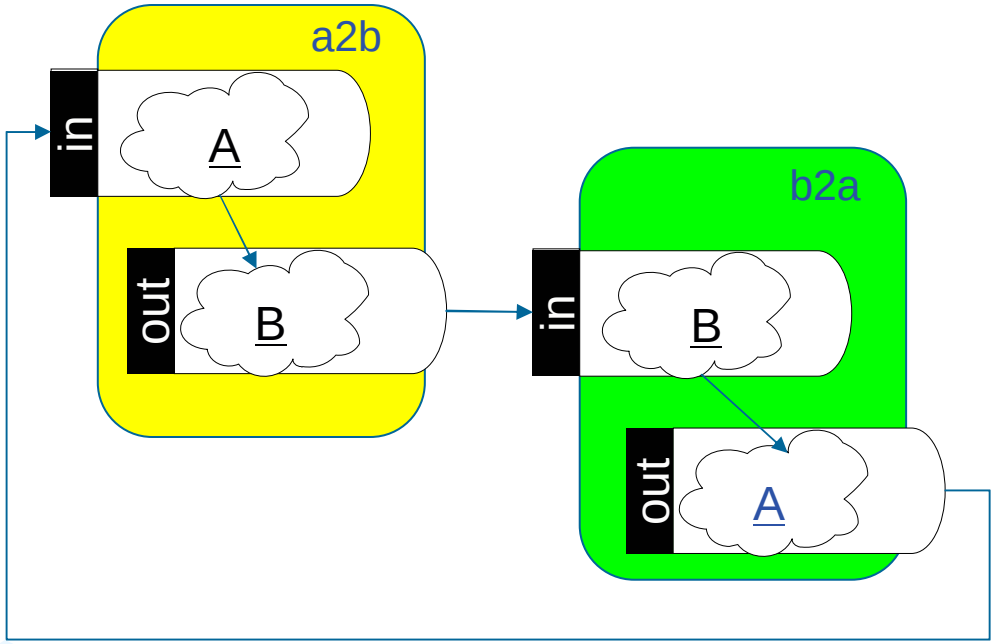
No ui_fixed


open.both
in
Widget
paint
Button
setLabel
out
Widget
setfont
setcolor
Button
open.color
in
Widget
paint
Button
setLabel
out
Widget
setcolor
Button
open.font
in
Widget
paint
Button
setLabel
out
Widget
setfont
Button

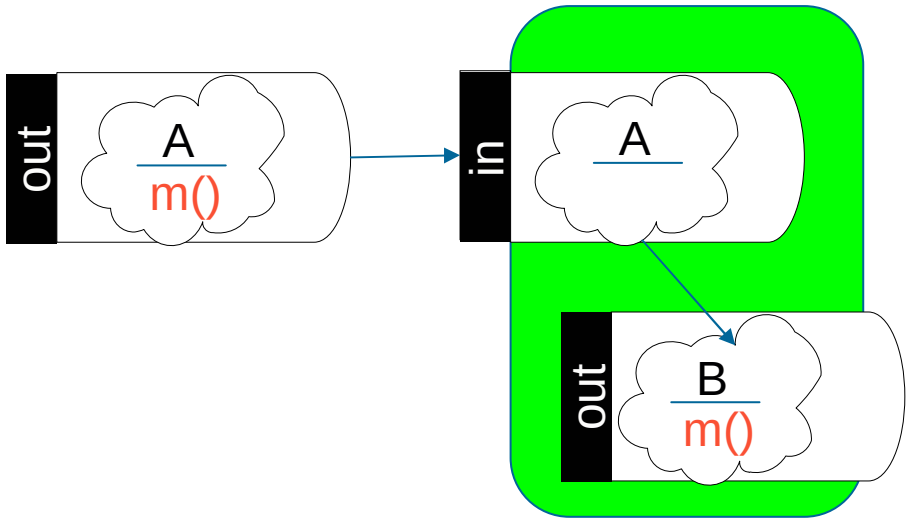
Type Checking

- class level type checking is done by Java compiler
- Unit-level type checking
  - Ensure atom's classes are consistent with atom's unit signature
  - Ensure the linking units within a compound is consistent
- Cyclic component linking have non-trivial effect on type checking
  - Inheritance cycles
  - Method collisions

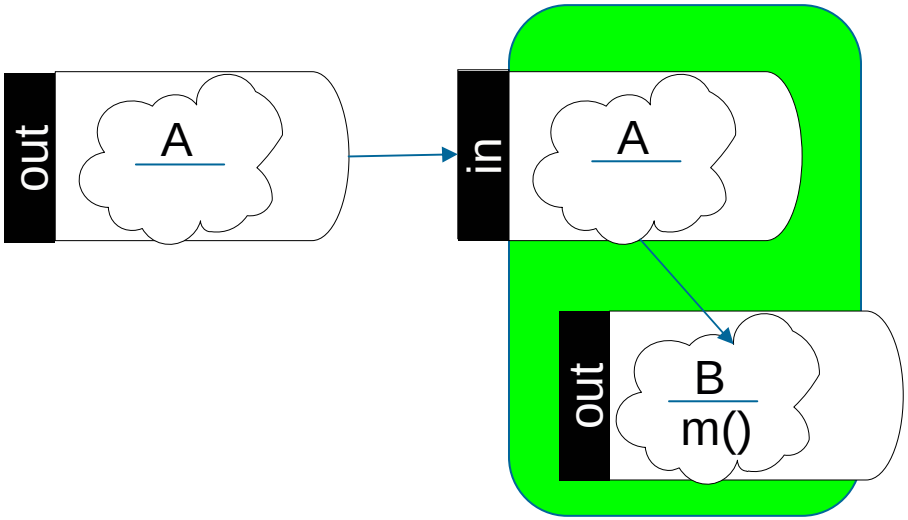
Inheritance cycles



Method collisions



Method collisions



Abstract Methods

- Java allows abstract methods to be unimplemented in concrete classes ?
- In Java, an abstract method invoked without an implementation will raise a runtime error.
- Jiazzi requires that concrete classes have no abstract methods
  - conflicts with Java's binary compatibility support.

Implementation

- A stub generator
- An offline linker
  - Unit-level type checking (the class level type checking is done by Java compiler)
  - Rewrites class files
    - Only class file's constant pools are rewritten (method bodies, debug attributes untouched)
    - Rename classes based on whether exported
    - Rename hidden methods to avoid collisions

Limitations

- No Native method
  - A Java native method is bound to a method based on the name of the method and its containing class.
- No Reflection
  - Symbols are referred to at runtime, while the Jizza linker works offline.

Conclusions

- Decoupling the package dependences
- Facilitating code reuse.
- Jiazzi does not change existing Java development practice: programs are still written in the Java language and still execute on conventional Java virtual machines.