

ReSwing – A Reactive Interface for Scala Swing

1 RESwing

Can we show the (complete?) interface of a RESwing component? I would expect there are some interesting things such as signals in the interface ecc.. – the complete interface can maybe get a bit large. Very small but representative parts of the interface are shown in Figure 5 and 6.

The RESwing library is an extension of the Scala Swing library, which wraps around Java Swing. The Scala Swing library mirrors the Java Swing class hierarchy and every component holds a reference to the underlying Java Swing component. Building on Scala Swing, the RESwing library adds another layer to this architecture. It provides its own class hierarchy containing all reactively enabled components. Figure 1 shows a small, representative part of these class hierarchies.

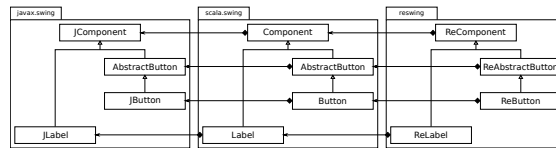


Figure 1: Scala Swing and ReSwing wrapper architecture

The RESwing library provides reactive values for certain properties of the Swing components. User code can provide signals to be used for these reactive values. As signals induce a highly declarative way of expressing the computation of values, they are not re-assignable once they are declared.

For that reason, reactive properties to be used with the RESwing library are passed to the components' constructors and cannot be assigned later. This also results in a slightly different approach when constructing ReSwing components as shown in Figure 2 and Figure 3. **I hope this is clearer now**

1.1 RESwing Events

The RESwing library provides REScala events for discrete changes, e.g., button clicks. Events exposed by RESwing components correspond to Scala Swing events, but integrate

```

1 val label = new Label
2 label.text = "foobar"
3 label.preferredSize = new Dimension(400, 40)

```

Figure 2: Label component instantiation in Scala Swing.

```

1 val label = new ReLabel(
2     text = "foobar"
3     preferredSize = new Dimension(400, 40)
4 )

```

Figure 3: ReLabel component instantiation in ReSwing.

with the REScala event system. This means all REScala event combinators and interface functions can be used with these events. Every RESwing event passes the triggering Swing event on. For example for a Swing `ButtonClicked` event, a REScala event of type `Event[ButtonClicked]` is fired.

But more expressive power ? What about the parameter type ? What about combinators?

1.2 RESwing Reactive Values

RESwing bridges between the Scala Swing getter, setter and reactor system and the REScala signal and event system. Clients can define reactive values that are passed to the RESwing interface. When these values change, the GUI interface is updated accordingly. Also, changes made by the user are reflected in the reactive values which the RESwing library provides.

Certain values can be changed by both the application and the user, i.e., they have multiple input sources. For example, the value representing the text in a text input field can be changed by the user by entering text, but it can also be set by the application. The library ensures consistency between values set by the application and those resulting from user interaction. This is achieved by disallowing the user to make changes in certain cases.

There are three different ways to initialize a reactive value. This aspect determines how changes to the reactive value are handled: **show also the changes of the value – I extended the text, I hope it is clearer now who can change which value when.**

- *Immediate value initialization* will set the reactive value to the value given by the client code immediately upon creation. The user running the application can potentially change the value afterwards by interacting with the user interface. The reactive value will automatically reflect any the changes made by the user when accessed from client code. **How? Can you show it in the example? – I labeled the example in Figure 4 appropriately**

- *Event initialization* will update the reactive value on each event occurrence. Client code can change the value by triggering an event in the stream with the new value. The user can also change the value by interacting with the user interface. The reactive value will automatically reflect any the changes made by the application user or client code.
- *Signal initialization* ensures that the reactive value always holds the value given by the signal. Hence, the application user is not allowed to change the value through the user interface.

For all these cases, the respective value is passed to the constructor of the component as shown in Figure 4.

```

1 // Immediate value initialization (using a string value)
2 val string: String
3 val label = new ReTextArea(
4     text = string
5 )
6
7 // Event initialization (using a string event value)
8 val event: Event[String]
9 val label = new ReTextArea(
10     text = event
11 )
12
13 // Signal initialization (using a string signal value)
14 val signal: Signal[String]
15 val label = new ReTextArea(
16     text = signal
17 )

```

Figure 4: Initializing a reactive value of a ReSwing component.

After a component instance has been constructed, all reactive values can be treated as signals when accessing the properties (e.g., `Signal { label.text() }`). Meaning no matter which of the methods described above was used to initialize the reactive value, it can be uniformly accessed and read as signal. **why not directly the signal? I would expect text is a signal – Yes, label.text is a signal (or more precisely can be implicitly casted to signal and thus just be used as signal), `Signal { label.text() }` was just an example**

1.3 Extending the RESwing Library by Reactive Values

The library offers a declarative syntax to define which reactive value should map to which property of the underlying Swing component. Using this syntax ensures that value changes are properly propagated from the ReSwing library to the Scala Swing library and vice versa.

For a reactive property, you need to specify:

- the *getter* of the underlying Swing property to retrieve the value

```

1 class ReLabel(val text: ReSwingValue[String] = ())
2     extends ReComponent {
3   text using (peer.text _, peer.text_ = _, "text")
4 }
5
6 class ReTextComponent(val text: ReSwingValue[String] = ())
7     extends ReComponent {
8   (text using (peer.text _, peer.text_ = _, classOf[ValueChanged])
9     force ("editable", peer.editable_ = _, false))
10 }
11
12 abstract class ReComponent extends ReUIElement {
13   val hasFocus = ReSwingValue using (peer.hasFocus _, classOf[FocusGained],
14                                     classOf[FocusLost])
15 }

```

Figure 5: Defining Reactive Values.

- the *setter* of the underlying Swing property to set the value (if the reactive property can be changed by client code, otherwise the reactive value will be read-only, i.e. changes by the user running the application will be reflected, but client code cannot directly change the value)
- a way to identify changes of the underlying Swing property, either by giving the bound property name or a Scala Swing event type

Additionally it is possible to force other properties to hold a specified value, if the reactive value should not be changeable by the user. This can be the case if the reactive value is initialized with a signal as described in section 1.2.

Examples of some reactive values defined in different ways are given in Figure 5.

1.4 Extending the RESwing Library by Events

Similar to defining RESwing values as outlined in section 1.3, it is also possible to

- define events that get fired when the underlying Swing component fires an event or
- declare events that offer client code the possibility to trigger certain actions by passing an event stream to the component.

Examples of these two use cases are given in Figure 6.

```
1 class ReButton extends ReComponent {  
2   val clicked = ReSwingEvent using classOf[ButtonClicked]  
3 }  
4  
5 class ReTextComponent(selectAll: ReSwingEvent[Unit] = ())  
6   extends ReComponent {  
7   selectAll using peer.selectAll _  
8 }
```

Figure 6: Defining Events.