ReSwing – A Reactive Interface for Scala Swing

1 RESwing

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.

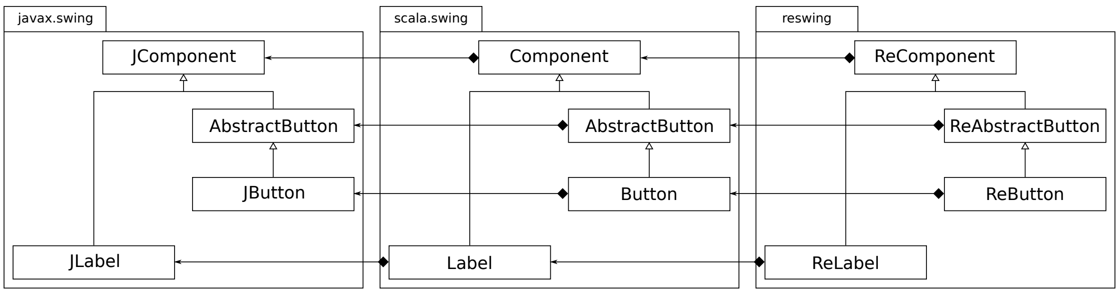

Figure 1: Scala Swing and ReSwing wrapper architecture

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.

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 inte-

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

Figure 2: Label component instantiation in Scala Swing.

```
val label = new ReLabel(
    text = "foobar"
    preferredSize = new Dimension(400, 40)
)
```

Figure 3: ReLabel component instantiation in ReSwing.

grate with the REScala event system. This means all REScala event combinators and interface functions can be used with these events. Every RESwing is associated to the respective Swing event, e.g., for a Swing `ButtonClicked` event a REScala event of type `Event[ButtonClicked]` is fired.

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.

ReSwing components expose immutable properties signals that user code can access. Note that, as in the rest of REScala, the signal is not reassignable, but the value it carries is, in the sense that it can change over time.

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.

Component properties are passed to the components' constructors and cannot be reassigned later. There are three different ways to initialize a reactive value. This aspect determines how changes to the reactive value are handled.

- *Immediate value initialization* will set the reactive value to the value given by the client code immediately upon creation. Client code cannot change this value directly, but the application user can change the value through the user interface if this is supported for the respective Swing property.

- *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.

- *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. An implicit conversion converts `A`, `Event[A]` and `Signal[A]` to `ReSwingValue[A]` to provide a uniform initialization approach.

```
// Immediate value initialization (using a string value)
val string: String = ...
val textArea = new ReTextArea(
    text = string
)

// textArea.text = "otherString" // Not possible

// Event initialization (using a string event value)
val event: Event[String] = ...
val textArea = new ReTextArea(
    text = event
)

// Signal initialization (using a string signal value)
val signal: Signal[String] = ...
val textArea = new ReTextArea(
    text = signal
)
```

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

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, developers need to specify:

- The *getter* of the underlying Swing property to retrieve the value.
- The *setter* of the underlying Swing property to set the value if the reactive property can be changed by client code. If no setter is specified, 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 providing the name of the bound property or by providing a Scala Swing event type. The event triggers a call to the getter to grab the most recent value.

Additionally, it is possible to force 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 reactive values defined in different ways are given in Figure 5.

```
class ReLabel(val text: ReSwingValue[String] = ())
                             extends ReComponent {
  text using (peer.text _, peer.text_= _, "text")
}

class ReTextComponent(val text: ReSwingValue[String] = ())
                            extends ReComponent {
  (text using (peer.text _, peer.text_= _, classOf[ValueChanged])
        force ("editable", peer.editable_= _, false))
}

abstract class ReComponent extends ReUIElement {
  val hasFocus = ReSwingValue using (peer.hasFocus _, classOf[FocusGained],
                                                      classOf[FocusLost])
}
```

Figure 5: Defining Reactive Values.

For properties exposed as events. Similar to the signal properties case, it is 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.

```
class ReButton extends ReComponent {
  val clicked = ReSwingEvent using classOf[ButtonClicked]
}

class ReTextComponent(selectAll: ReSwingEvent[Unit] = ())
                            extends ReComponent {
  selectAll using peer.selectAll _
}
```

Figure 6: Defining Events.