ReSwing – A Reactive Interface for Scala Swing

1 Basic Design Overview – Java Swing / Scala Swing / ReSwing

The Scala Swing library is implemented as a wrapper around the Java Swing library. It 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.

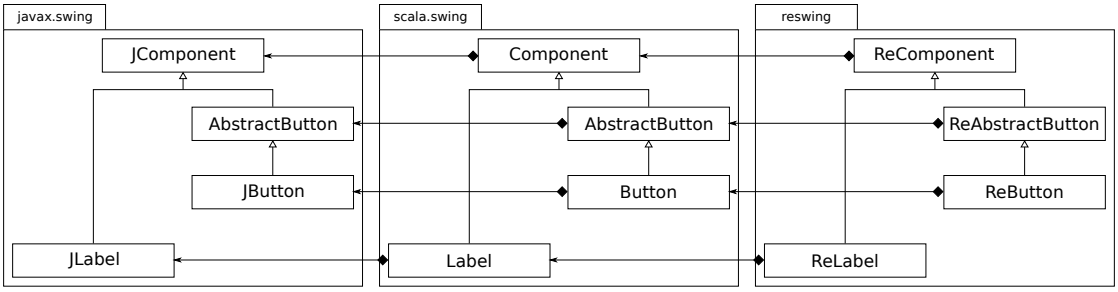

Figure 1: Scala Swing and ReSwing wrapper architecture

The purpose of the ReSwing library is to provide reactive values for certain Swing properties. Since it is possible to define a signal for these values and signals are defined once the signal is created and are not re-assignable, properties in 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 listing 1 and listing 2.

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

Listing 1: Example for a Label component instantiation in Scala Swing

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

Listing 2: Example for a ReLabel component instantiation in ReSwing

2 ReSwing Events

The ReSwing library provides ReScala events for Scala Swing events like, e.g., button clicks. They can be accessed and used just like any other ReScala event. Every occurrence of a Scala Swing event causes the related ReScala event of the ReSwing component to be fired.

3 ReSwing Reactive Values

The ReSwing library allows the application to set reactive values and will then ensure that the underlying Swing library updates the user interface accordingly. Also, changes made by the user that are published by the Scala Swing library are reflected in the reactive values which the ReSwing library provides. So it acts as a middleware to translate between the Scala Swing getter, setter and reactor system and the ReScala signal and event system.

There are values (e.g. the value representing the text in a text input field) that can be changed by both the application and the user. This rises the question of how to handle two different input sources for these reactive values. The library ensures that the signals set by the application and the changes resulting from user interaction are consistently represented. This is achieved by disallowing the user to make changes in certain cases.

There are three different ways to initialize a reactive value which determine how changes to reactive values are handled:

- *Initializing directly with the value* will set the reactive value to the given value immediately upon creation. The user can change the value afterwards.
- *Initializing with an event stream* will update the reactive value on each event occurrence. The user can change the value.
- *Initializing with a signal* ensures that the reactive value always holds the value given by the signal. Hence, the user is not allowed to change the value.

For all three cases, the respective value can just be passed to the constructor of the component as shown in listing 3.

```
val value: String
val label = new ReTextArea(
    text = value
)
```

```
val event: Event[String]
val label = new ReTextArea(
    text = event
)

val signal: Signal[String]
val label = new ReTextArea(
    text = signal
)
```

Listing 3: 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. in `Signal { ... }` expressions).

4 Extending the ReSwing Library – Defining 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
- the *setter* of the underlying Swing property to set the value (if the reactive property can be changed by the application)
- 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 3.

Examples of some reactive values defined in different ways are given in listing 4.

```
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[Boolean] = ()
  hasFocus using (peer.hasFocus _, classOf[FocusGained]),
                                   classOf[FocusLost])
}
```

Listing 4: Defining Reactive Values