CHAPTER 1

Reusing and composing elements

A key design goal of Spec is to enable the seamless reuse of user interfaces as widgets for the user interface you are building. The reason for this is that it results in a significant productivity boost when creating user interfaces.

This focus on reuse was actually already visible in the previous chapter, where we have seen that basic widgets can be used as if they were a complete user interface. In this section we focus on the reuse and composition of widgets, showing that it basically comes for free. The only requirement when building a UI is to consider how the user interface should be parameterized when it is being reused, and there is only one concrete rule that needs to be followed in that respect.

In this chapter, you will learn how we can build a new UI by reusing already defined elements.

1.1 First requirements

To show how Spec enables the composition and reuse of user interfaces, in this chapter we build the user interface shown in Figure 1.1 as a composition of four parts:

1. The **WidgetClassList**: a widget containing a `ListModel` specifically for displaying the subclasses of `AbstractWidgetModel`.
2. The **ProtocolMethodList**: a widget composed of a `ListModel` and a `LabelModel` for displaying methods of a protocol.

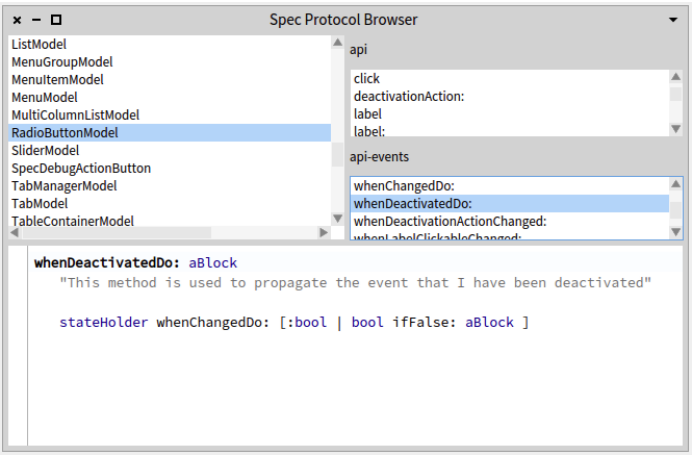

Figure 1.1 Screen shot of the ProtocolBrowser.

3. The **ProtocolViewer**: a composition of one `WidgetClassList` and two `ProtocolMethodList`, it will browse the methods in the protocols `api` and `api-events` of all subclasses of `AbstractWidgetModel`.
4. The **ProtocolBrowser** : reuses a `ProtocolViewer`, changes its layout and adds a `TextModel` to see the source code of the methods.

1.2 Creating a basic UI to be reused as a widget

The first custom UI we build should display a list of all subclasses of `AbstractWidgetModel`. This UI will later be reused as a widget for a more complete UI. The code is as follows (we do not include code for accessors):

First we create a subclass of `ComposableModel` with one instance variable `list` which will hold an instance of `ListModel`.

```
ComposableModel subclass: #WidgetClassList
  instanceVariableNames: 'list'
  classVariableNames: ''
  package: 'Spec-BuildUIWithSpec
```

In the method `initializeWidgets`, we create the list and populate it with the required classes, in alphabetical order. We also add a title for the window.

```
WidgetClassList >> initializeWidgets
  list := self newList.
  list items: (AbstractWidgetModel allSubclasses
                  sorted: [:a :b | a name < b name ]).
  self focusOrder add: list.
```

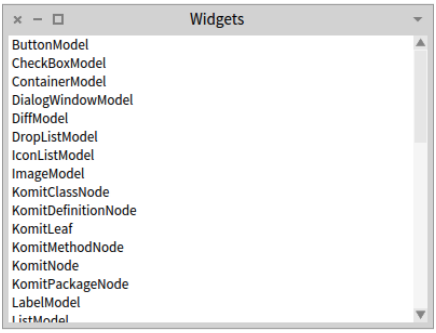

Figure 1.2 Screen shot of the WidgetClassList.

```
WidgetClassList >> title
  ^'Widgets'
```

The layout contains only the list:

```
WidgetClassList class >> defaultSpec
  ^ SpecLayout composed
    add: #list;
    yourself
```

The resulting UI is shown in Figure 1.2.

Since this UI will later be used together with other widgets to provide a more complete user interface, some actions will need to occur when a list item is clicked. However, we cannot know beforehand what all these possible actions will be everywhere that it will be reused. The best solution therefore is to place this responsibility on the reuser of the widget. Every time this UI is reused as a widget, it will be configured by the reuser. To allow this, we add a configuration method namd `wheSelectedItemChanged:`, in the `api` protocol:

```
WidgetClassList >> whenSelectedItemChanged: aBlock
  list whenSelectedItemChanged: aBlock
```

Now, whoever reuses this widget can parameterize it with a block that will be executed whenever the selected item is changed.

> **Note** The only rule for reuse of Spec widgets is that all configuration methods of your UI should be contained in a `api` protocol. This is to make it easier for reusers of this widget to discover how it can be parameterized.

To do are we sure that we want API and that we want to force to define such method.

1.3 Combining two basic widgets into a reusable UI

The UI we build now will show a list of all methods of a given protocol, and it combines two widgets: a list and a label. Considering reuse, there is no difference with the previous UI. This is because the reuse of a UI as a widget is **not impacted at all** by the number of widgets it contains (nor by their position). Large and complex UIs are reused in the same way as simple widgets.

```
ComposableModel subclass: #ProtocolMethodList
  instanceVariableNames: 'label methods
  classVariableNames: ''
  package: 'Spec-BuildUIWithSpec'
```

The `initializeWidgets` method for this UI is quite straightforward. We specify the default label text as 'protocol', which will be changed when the widget is reused. We also give this UI a title.

```
ProtocolMethodList >> initializeWidgets
  methods := self newList.
  methods displayBlock: [ :m | m selector ].
  label :=  self newLabel.
  label label: 'Protocol'.
  self focusOrder add: methods.
```

```
ProtocolMethodList >> title
  ^ 'Protocol widget'
```

The layout code builds a column with the fixed-height label on top and the list taking all the space that remains. (See Chapter **??** for more layout information).

```
ProtocolMethodList class >> defaultSpec
  ^ SpecColumnLayout composed
      add: #label height: self toolbarHeight;
      add: #methods;
      yourself
```

This UI can be seen by evaluating `ProtocolMethodList new openWithSpec`. As shown in Figure 1.3 the list is empty. This is normal since we did not set any items.

Our protocol method list will need to be configured when it is used, for example to fill the list of methods and to specify what the name is of the protocol. To allow this, we add a number of configuration methods in the `api` protocol:

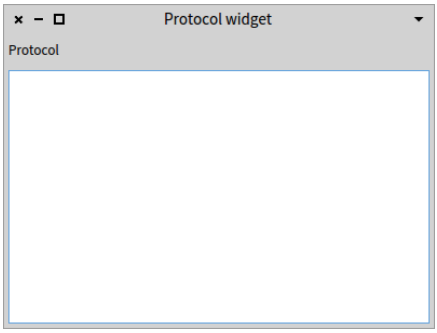

Figure 1.3 Screen shot of the ProtocolMethodList.

```
ProtocolMethodList >> items: aCollection
  methods items: aCollection
```

```
ProtocolMethodList >> label: aText
  label label: aText
```

```
ProtocolMethodList >> resetSelection
  methods resetSelection
```

```
ProtocolMethodList >> whenSelectedItemChanged: aBlock
  methods whenSelectedItemChanged: aBlock
```

To do Johan are we sure that we want to promote such dummy methods?

1.4 Managing three widgets and their interactions

The third user interface we build is a composition of the two previous user interfaces. We will see that there is no difference between configuring custom UIs and configuring system widgets: both kinds of widgets are configured by calling methods of the `api` protocol.

This UI is composed of a `WidgetClassList` and two `ProtocolMethodList` and specifies that when a model class is selected in the `WidgetClassList`, the methods in the protocols `api` and `api-events` will be shown in the two `ProtocolMethodList` widgets.

```
ComposableModel subclass: #ProtocolViewer
  instanceVariableNames: 'models api events'
  classVariableNames: ''
  package: 'Spec-BuildUIWithSpec'
```

The `initializeWidgets` method shows the use of `instantiate:` to instantiate widgets, and some of the different parametrization methods of the `ProtocolMethodList` class.

```
ProtocolViewer >> initializeWidgets
  models := self instantiate: WidgetClassList.
  api := self instantiate: ProtocolMethodList.
  events := self instantiate: ProtocolMethodList.

  api label: 'api'.
  events label: 'api-events'.

  self focusOrder add: models; add: api; add: events.
```

```
ProtocolViewer >> title
  ^ 'Protocol viewer'
```

To describe the interactions between the different widgets we define the `initializePresenter` method. It specifies that when a class is selected, the selections in the method lists are reset and both method lists are populated. Additionally, when a method is selected in one method list, the selection in the other list is reset.

```
ProtocolViewer >> initializePresenter
  models whenSelectedItemChanged: [ :class |
    api resetSelection.
    events resetSelection.
    class
      ifNil: [ api items: #(). events items: #() ]
      ifNotNil: [
        api items: (self methodsIn: class for: 'api').
        events items: (self methodsIn: class for: 'api-events') ] ].

  api whenSelectedItemChanged: [ :method |
    method ifNotNil: [ events resetSelection ] ].
  events whenSelectedItemChanged: [ :method |
    method ifNotNil: [ api resetSelection ] ].
```

```
ProtocolViewer >> methodsIn: class for: protocol
  ^ (class methodsInProtocol: protocol) sorted:
        [ :a :b | a selector < b selector ].
```

Lastly, the layout puts the sub widgets in one column, with all sub widgets taking the same amount of space.

```
ProtocolViewer class >> defaultSpec
  ^ SpecColumnLayout composed
      add: #models; add: #api; add: #events;
      yourself
```

As previously, the result can be seen by executing the following snippet of code: `ProtocolViewer new openWithSpec`, and the result is shown in Figure 1.4. This user interface is functional, clicking on a class will show the methods of the `api` and the `api-events` protocols of that class.

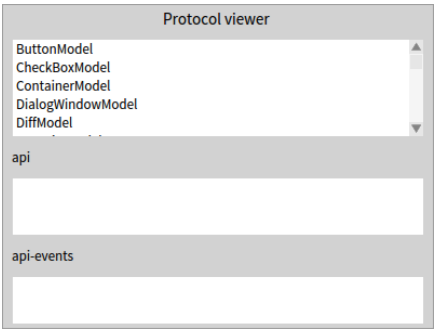

Figure 1.4 Render of the ProtocolViewer.

Similar to the second user interface, when this UI is reused it will probably need to be configured. The relevant configuration here is what to do when a selection change happens in any of the three lists. We hence add the following three methods to the `api` protocol.

```
ProtocolViewer >> whenClassChanged: aBlock
  models whenSelectedItemChanged: aBlock
```

```
ProtocolViewer >> whenEventChanged: aBlock
  events whenSelectedItemChanged: aBlock
```

```
ProtocolViewer >> whenAPIChanged: aBlock
  api whenSelectedItemChanged: aBlock
```

1.5 Changing the layout of a reused widget

Sometimes, when you want to reuse an existing UI as a widget, the layout of that UI is not appropriate to your needs. Spec allows you to nonetheless reuse such a UI by overriding the layout of its widgets, and we show this here.

Our last user interface reuses the `ProtocolViewer` with a different layout and adds a text zone to edit the source code of the selected method.

```
ComposableModel subclass: #ProtocolBrowser
  instanceVariableNames: 'text viewer'
  classVariableNames: ''
  package: 'Spec-BuildUIWithSpec'
```

```
ProtocolBrowser >> initializeWidgets
  text := self instantiate: TextModel.
  viewer := self instantiate: ProtocolViewer.
  text
    aboutToStyle: true;
    isCodeCompletionAllowed: true.
```

```
  self focusOrder
    add: viewer;
    add: text.
```

```
ProtocolBrowser >> title
  ^'Spec Protocol Browser'
```

The text field is configured to show source code:

- `aboutToStyle: true` enables syntax highlighting.
- `isCodeCompletionAllowed: true` enables code completion.

The `initalizePresenter` method is used to make the text zone react to a selection in the lists. When a method is selected, the text zone updates its contents to show the source code of the selected method.

```
ProtocolBrowser >> initializePresenter
  viewer whenClassChanged: [ :class | text behavior: class ].
  viewer whenAPIChanged: [ :item |
    item
      ifNil: [ text text: '' ]
      ifNotNil: [ text text: item sourceCode ] ].
  viewer whenEventChanged: [ :item |
    item
      ifNil: [ text text: '' ]
      ifNotNil: [ text text: item sourceCode ] ]
```

The last piece of the puzzle is the layout of the different widgets that we are reusing. We combine columns and rows in this UI. The first row is special because we reuse the internal widgets of the `viewer` widget in a different layout. To do this, we specify the sequence of accessor messages that need to be sent (as an array of symbols): for example, for the list of classes first the `viewer` message and then the `models` message.

```
ProtocolBrowser class >> defaultSpec
  ^ SpecLayout composed newColumn: [:col |
      col newRow: [ :row |
        row add: #(viewer models);
            newColumn: [ :col2 |
              col2 add: #(viewer api);
                   add: #(viewer events) ] ];
            add: #text];
      yourself
```

> **Note** To layout internal widgets of a widget you are reusing (instead of the widget in its entirety), give an array of symbols: it states the sequence of accessors that need to be sent to get to the internal widget.

This concludes the last example of this chapter, and the result can be seen in the first figure of this chapter, Figure 1.1.

1.6 Conclusion

In this chapter we have discussed a key point of Spec: the ability to seamlessly reuse existing UIs as widgets. This ability comes with no significant cost to the creator of a UI. The only thing that needs to be taken into account is how a UI can (or should) be customized, and these customization methods should be placed in a `api` protocol for discoverability.

The reuse of complex widgets at no significant cost was a key design goal of Spec because it is an important productivity boost for the writing process of UIs. The boost firstly comes from being able to reuse existing nontrivial widgets, and secondly because it allows you to structure your UI in coherent and more easily manageable sub-parts with clear interfaces. We therefore encourage you to think of your UI as a composition of such sub-parts and construct it modularly, to yield greater productivity.