CHAPTER 1

The fundamentals of Spec

In this chapter we revisit the key aspects of Spec and put in perspective the important customization points of its building process.

1.1 User interface building: a composition

A key aspect of Spec is that all user interfaces are constructed through the reuse and composition of existing user interfaces. To allow this, defining a user interface consists of defining the *model* of the user interface, and *not* the user interface elements that will be shown on screen. These UI elements are instantiated by Spec, taking into account the underlying UI framework.

In the end, it is the composition of the model and the UI elements that makes up the resulting user interface that is shown. This also explains the name of the root component of Spec: since all UIs are constructed through *composition* of other UI's, and it is sufficient to define the *model* to define the UI, the root class of all UIs is named `ComposableModel`. Consequently, to define a new user interface, a subclass of `ComposableModel` needs to be created.

Considering the construction and orchestration of the different widgets in a user interface, Spec is inspired by the Model-View-Presenter pattern. In Spec, the model is actually a presenter in the MVP triad. Note that future versions of Spec may rename the `ComposableModel` into `ComposablePresenter` to convey such situation.

Fundamentally, it is built around three concerns that materialize themselves as the following three methods in `ComposableModel`:

- `initializeWidgets` treats the widgets themselves
- `initializePresenter` treats the interactions between widgets

- `defaultSpec` treats the layout of the widgets

These methods are hence typically found in the model for each user interface. In this chapter we describe the finer points of each method and how these three work together to build the overall UI.

1.2 The *initializeWidgets* method

The method `initializeWidgets` instantiates, saves in instance variables, and partially configures the different widgets that will be part of the UI. Instantiation of the models will cause the instantiation and initialization of the different lower-level user interface components, constructing the UI that is shown to the user. A first part of the configuration of each widget is specified here as well, which is why this method is called `initializeWidgets`. This focus in this method is to specify what the widgets will look like and what their self-contained behavior is. The behavior to update model state, e.g., when pressing a `Save` button, is described in this method as well. It is explicitly *not* the responsibility of this method to define the interactions *between* the widgets.

In general the `initializeWidgets` method should follow the pattern:

- widget instantiation
- widget configuration specification
- specification of focus order

The last step is not mandatory but *highly* recommended. Indeed, without this final step keyboard navigation will not work reliably.

> **Note** Specifying the method `initializeWidgets` is mandatory, as without it the UI would have no widgets.

Widget instantiation

The instantiation of the model for a widget can be done in two ways: through the use of a creation method or through the use of the `instantiate:` method.

- Considering the first option, the framework provides unary messages for the creation of all basic widgets. The format of these messages is `new[Widget]`, for example `newButton` creates a button widget, and `newList` creates a list widget. The complete list of available widget creation methods can be found in the class `ComposableModel` in the protocol `widgets`.
- The second option is more general: to reuse a `ComposableModel` subclass (other than the ones handled by the first option) the widget needs to be instantiated using the `instantiate:` method. For example, to

reuse a `MessageBrowser` widget, the code is `self instantiate: MessageBrowser.`

1.3 The *layout* method

> **Note** may be the title should be Expressing UI Layouts and the first sentence should be. Widget layout is expressed defining methods that specify how the different widgets composing a UI are placed. As we will such methods can have different name but should be tagged to the recognized by the system.

The method `layout` specifies the layout of the different widgets in the UI. It also specifies how a widget reacts when the window is resized. The layout method is placed at the class side because it typically returns a value that is the same for all the instances. Put differently, typically all the instances of the same user interface have the same layout and hence this can be considered as being a class-side accessor for a class variable.

> **Note** Specifying the method `layout` is mandatory, as without it the UI would show no widgets to the user.

> **To do** johan this is strange because we say that the layout method is mandatory but none of the example explicitly defines one. Even the lookup does not look for the layout method. So we should rephrase this. I did not check the image to check what to write but to me. May be we should say layout methods (methods defining spec layout objects) not sure that it is working? May be this is the first sentence that is misleading.

Having multiple layouts for a widget

For the same UI, multiple layouts can be described, and when the UI is built the use of a specific layout can be indicated. To do this, instead of calling `openWithSpec` (as we have done until now), use the `openWithSpec:` message with the name of the layout method as argument. For example, consider the following artificial example of a two button UI that has two different layout methods:

```
ComposableModel subclass: #TwoButtons
  instanceVariableNames: 'button1 button2'
  classVariableNames: ''
  package: 'Spec-BuildUIWithSpec'
```

```
TwoButtons >> initializeWidgets
  button1 := self newButton.
  button2 := self newButton.
```

```
  button1 label: '1'.
  button2 label: '2'.

  self focusOrder
    add: button1;
    add: button2
```

```
TwoButtons class >> buttonRow
  <spec: #default>
  ^SpecRowLayout composed
    add: #button1; add: #button2;
    yourself
```

```
TwoButtons class >> buttonCol
  ^SpecColumnLayout composed
    add: #button1; add: #button2;
    yourself
```

This UI can be opened in multiple ways:

- `TwoButtons new openWithSpec: #buttonRow` places the buttons in a row.
- `TwoButtons new openWithSpec: #buttonCol` places them in a column.

Note that the `buttonRow` layout method has a `<spec: #default>` pragma. As a result, the `buttonRow` layout will be used when executing `TwoButtons new openWithSpec`, as we explain next.

The `openWithSpec` method uses the following lookup mechanism to obtain the layout method:

1. Search on class side, throughout the whole class hierarchy, for a method with the pragma `<spec: #default>`.
2. If multiple such methods exist, the first one that is found is used.
3. If no such methods exist and if there is exactly one method with the pragma `<spec>`, this method is used.
4. Otherwise the class-side `defaultSpec` method will be called, which will raise `subclassResponsibility` if it is not overriden.

> **Note** If there is a method with the `<spec: #default>` or `<spec>` pragma in a superclass of the UI you are building, this one will be used instead of a `defaultSpec` method implemented in your class.

For example, subclasses of `TwoButtons` cannot use `defaultSpec` methods since there is a method with the `<spec: #default>` pragma.

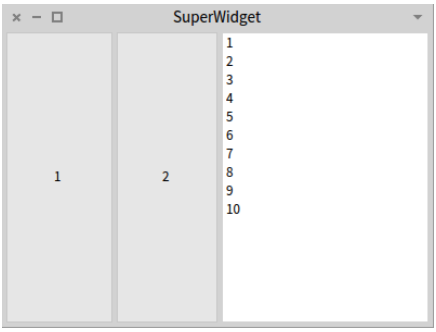

Figure 1.1 Screen shot of the UI with buttons placed horizontally

Specifying a layout when reusing a widget

Having multiple layouts for a widget implies that there is a way to specify the layout to use when a widget is reused.

Until now, we have used the `add:` message to add a widget to a layout. This uses the mechanism of `openWithSpec` to determine the layout of the widget that is being reused. To use an alternative layout for the widget that is being reused, use the `add:withSpec:` message. It takes as extra argument the name of the layout method to use, as we show below:

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

```
TBAndListH >> initializeWidgets
  buttons := self instantiate: TwoButtons.
  list := self newList.
  list items: (1 to: 10).
  self focusOrder add: buttons; add: list.
```

```
TBAndListH >> title
  ^'SuperWidget'
```

```
TBAndListH class >> defaultSpec
  ^ SpecRowLayout composed
      add: #buttons; add: #list;
      yourself
```

This `TBAndListH` class results in a **SuperWidget** window as shown in Figure 1.1. It reuses the `TwoButton` widget, and places all three widgets in a horizontal order because the `TwoButton` widget will use the `buttonRow` layout method.

Alternatively, we can create `TBAndListV` class as a subclass of `TBAndListH` and only change the `defaultSpec` method as below. It specifies that the

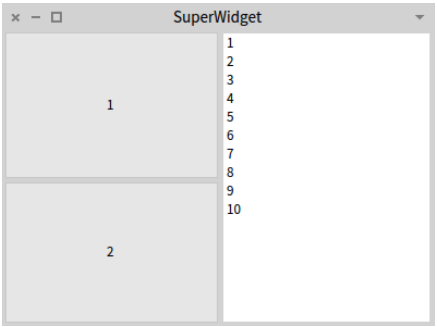

Figure 1.2 Screen shot of the UI with buttons placed vertically

reused `buttons` widget should use the `buttonCol` layout method, and hence results in the window shown in Figure 1.2.

```
TBAndListH subclass: #TBAndListV
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Spec-BuildUIWithSpec'
```

```
TBAndListV class >> defaultSpec
  ^ SpecRowLayout composed
      add: #buttons withSpec: #buttonCol; add: #list;
      yourself
```

1.4 The *initializePresenter* method

The method `initializePresenter` defines the interactions between the different widgets. By connecting the behaviors of the different widgets it specifies the overall presentation, i.e., how the overall UI responds to interactions by the user. Usually this method consists of specifications of actions to perform when a certain event is received by a widget. The whole interaction flow of the UI then emerges from the propagation of those events.

> **Note** The method `initializePresenter` is the only optional method for a Spec UI.

In Spec, the different UI models are contained in value holders, and the event mechanism relies on the announcements of these value holders to manage the interactions between widgets. Value holders provide the method `whenChangedDo:` that is used to register a block to perform on change and the method `whenChangedSend: aSelector to: aReceiver` to send a meessage to a given object. In addition to these primitives methods, the basic widgets provide more specific hooks, e.g., when an item in a list is selected or deselected.

1.5 Conclusion

In this chapter we have given a more detailed description of how the three fundamental methods of Spec: `initializeWidgets`, `defaultSpec` and `initializePresenter` are each responsible for a different aspect of the user interface building process. Notably, we also discussed in detail the ability to use different layout methods and how the lookup of layout methods is performed.

> **Note** Although reuse is fundamental in Spec, we did not explicitly treat it in this chapter. Instead we refer to the previous chapter for more information.