The Spec UI framework

Johan Fabry, Stéphane Ducasse

The Spec UI framework

Johan Fabry and Stéphane Ducasse

February 6, 2017

[&] Published by Square Bracket Associates, Switzerland.
http://squarebracketassociates.org

ISBN 978-1-326-92746-2
First Edition, January 2017.

Layout and typography based on the sbabook LaTeX class by Damien Pollet. The source code of the book itself lives at https://github.com/SquareBracketAssociates/BuildingUIWithSpec

Contents

Illustrations

CHAPTER 1

Introduction

Spec is a framework in Pharo for describing user interfaces. It allows for the construction of a wide variety of UIs; from small windows with a few buttons up to complex tools like a debugger. Indeed multiple tools in Pharo are written in Spec, e.g., the Watchpoint window, Komitter, Change Sorter, Critics Browser, and the Pharo 3 debugger.

The fundamental principle behind Spec is reuse of user interface logic and visual composition. User interfaces are built by reusing and composing existing user interfaces, configuring them as needed. This principle starts from the most primitive elements of the UI: widgets such as buttons and labels are in themselves complete UIs that can be reused, configured, and opened in their own window. These elements can be combined to form more complex UIs that again can be reused as part of a bigger UI, and so on. This is somewhat similar to how the different tiles on the cover of this book are combined. Smaller tiles configured with different colors or patterns join to form bigger rectangular shapes that are a part of an even bigger floor design.

To allow such reuse, Spec is influenced by VisualWorks and Dolphin Smalltalks' Model View Presenter (MVP) pattern. Spec recognizes the need for a Presenter or ApplicationModel class (in Spec called ComposableModel) that manages the logic and the link between widgets and domain objects. Fundamentally, when writing Spec code, the developer does *not* come into contact with UI widgets, instead a ComposableModel is programmed that holds the UI logic. When the UI is opened this model will then instantiate the appropriate widgets. This being said, for the programmer this distinction is not really apparent and it feels as if the widgets are being programmed directly.

Spec is a standard UI framework in Pharo and differs from the other UI frameworks present: Morphic and the GT Tools. Morphic is a general-purpose graphics system originally built for the Self language, and later included in

Squeak (the ancestor of Pharo). Morphic provides standard UI widgets, and as such is used by Spec to render UI's, but can also be used to build any kind of graphical shape and allows for their direct manipulation. Spec is more restricted than Morphic in that it only allows one to build user interfaces for applications that have typical GUI widgets such as buttons, lists et cetera. The GT tools are more restrictive than Spec since they target the domain of development tools. The GT Tools therefore restrict and simplify UI construction to layouts and workflows typical to development environments. Spec does not have these restrictions, which can make UI construction more intricate but allows for a wider variety of UIs to be built.

A significant drawback of Spec until now is that it did not have good quality documentation. Documentation was scattered, sometimes hard to understand and overall not user friendly. We believe that this has caused unneeded barriers to adoption and have therefore written this book to adress this issue. While the contents of this book is based on existing Spec documentation, almost all text has been completely rewritten in order to provide better quality documentation. We hope that this book will be of use to developers that need to write UI's in Pharo by significantly easing the UI development experience with Spec.

> **Note** While Spec comes as standard in Pharo 3 and above, this book focuses on Pharo 5 and 6. Earlier and later versions of Pharo may have implementations of Spec that differ, causing some example code here to not work. Nonetheless, the fundamental principles of UI development in Spec will hold and example code changes will therefore be minor.

This book is meant to be read as follows: Chapter 2 and 3 give a first contact with Spec and talk about how reuse is at the core of Spec. These chapters should be read completely by a Spec novice. The fourth chapter treats the fundamentals of Spec and gives a more complete, conceptual overview of how the different parts of a Spec UI work together. This is recommended reading for all Spec users, since a better understanding of the fundamentals will ease UI development at all user experience levels. Chapters 5 and beyond are considered more as reference material to be read on demand. This being said, Chapter 5 treats layouts, which is required by all UIs. Hence it does make sense for all Spec users to read it, so that they can construct their UI layout in the best possible way. Lastly, Chapter 9 gives an assortment of tips and tricks that can be useful in a wide variety of settings, so we recommend all readers of this book to at least browse through it.

CHAPTER 2

First Contact With Examples

To start this booklet we use a couple of examples to explain the use of Spec. We will first construct a small but complete user interface, and then show some more examples of how existing widgets can be configured. This will already allow you to build basic user interfaces.

After completing this chapter you should at least read the following chapter about reuse of Spec widgets, which is the key reason behind the power of Spec. With these two chapters read you should be able to construct Spec user interfaces as intended. You could use the rest of this booklet just as reference material, but nonetheless we recommend you to at least give a brief look at the other chapters as well.

2.1 A customer satisfaction UI

In this first example of a Spec UI we will construct a simple customer satisfaction UI, which will allow a user to give feedback about a service by clicking on one of three buttons. (This feedback should be recorded and processed, but this is outside of the scope of this text). We show a screenshot of the UI in Figure 2-1.

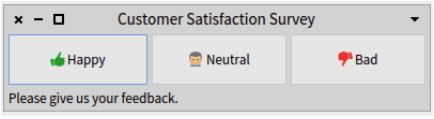

Figure 2-1 A screen shot of the customer satisfaction UI.

Create the class of the UI and variable accessors

All user interfaces in Spec are subclasses of `ComposableModel`, so the first step in creating the UI is creating a subclass:

```
ComposableModel subclass: #CustomerSatisfaction
  instanceVariableNames: 'buttonHappy buttonNeutral buttonBad screen'
  classVariableNames: ''
  package: 'Spec-BuildUIWithSpec'
```

The instance variables of the class hold the widgets that the UI contains. In this case we have three buttons and a text screen. Accessors for these instance variables also need to be defined, the Spec interpreter will use them in the UI construction process.

> **Note** Always remember to generate the accessors for the instance variables of the widgets of the UI.

The methods of the class provide their initialization and configuration, e.g. labels and actions, as well as the logic of their interaction. The basic design of our GUI, i.e. how the widgets are laid out, is defined by a method at class side.

Instantiate and configure subwidgets

A subclass of `ComposableModel` has the responsibility to define the `initializeWidgets` method, which instantiates and configures the widgets used in the user interface. We now discuss it piece by piece.

First we show widget instantiation:

```
CustomerSatisfaction >> initializeWidgets

  "widget instantiation"
  screen := self newLabel.
  buttonHappy := self newButton.
  buttonNeutral := self newButton.
  buttonBad := self newButton.
```

`ComposableModel` defines messages for the creation of the standard widgets: `newButton`, `newCheckBox`, `newDropList`, ... All of these are defined in the `widgets` protocol.

> **Note** **Do not** call `new` to instantiate a widget that is part of your UI.

> **Note** An alternative way to instantiate widgets is by using the message `instantiate:` with a class as argument. For example `screen := self instantiate: LabelModel.` This allows one to instantiate non-standard widgets, but can of course also be used for the standard widgets.

Second, we configure the buttons of our UI. The message `label:` defines their label and the message `icon:` specifies the icon that will be displayed near the label. The code below is for Pharo 6, for this to work in Pharo 5 replace `self iconNamed:` with `Smalltalk ui icons iconNamed:`.

```
[ ... continued ... ]
  "widget configuration"
  screen label: 'Please give us your feedback.'.
  buttonHappy
    label: 'Happy';
    icon: (self iconNamed: #thumbsUp).
  buttonNeutral
    label: 'Neutral';
    icon: (self iconNamed: #user).
  buttonBad
    label: 'Bad';
    icon: (self iconNamed: #thumbsDown).
```

Third and last, it is a good practice to define the focus order, which is useful for keyboard navigation.

```
[ ... continued ... ]
  "specification of order of focus"
  self focusOrder
    add: buttonHappy;
    add: buttonNeutral;
    add: buttonBad
```

A `ComposableModel` subclass may also define an `initializePresenter` method. The purpose of this method is to configure the interactions between the different widgets.

```
CustomerSatisfaction >> initializePresenter

  buttonHappy action: [ screen label: buttonHappy label ].
  buttonNeutral action: [ screen label: buttonNeutral label ].
  buttonBad action: [ screen label: buttonBad label ].
```

We use the message `action:` to specify the action that is performed when the buttons are clicked. In this case, we change the content of what is shown on the screen, to provide feedback that the choice has been registered. Note that the message `action:` is part of the button API. In other situations, you will specify that when a given event occurs, another message should be sent to a widget subpart.

> **Note** To summarize: the configuration of a widget 'on its own' goes in `initializeWidgets` and the configuration of a widget 'in cooperation with others' goes in `initializePresenter`.

The widgets have now been defined and configured, but their placement in

the UI has not yet been specified. This is the role of the class side method `defaultSpec`.

```
CustomerSatisfaction class >> defaultSpec
  ^ SpecLayout composed
    newRow: [ :row |
      row add: #buttonHappy; add: #buttonNeutral; add: #buttonBad ]
    origin: 0 @ 0 corner: 1 @ 0.7;
    newRow: [ :row | row add: #screen ]
    origin: 0 @ 0.7 corner: 1 @ 1;
    yourself
```

In this layout, we add two rows to the UI, one with the buttons and one with the screen of text. Defining widget layout is a complex process with many different possible requirements, hence in this chapter we do not talk in detail about layout specification. Instead for more information we refer to Chapter 5.

> **Note** The argument of the `add:` messages are the symbols for the accessors. This is because the Spec interpreter will call these methods to retrieve the widgets when performing the layout.

Define a title and window size, open and close the UI

To set the window title and the initial size, two more methods need to be defined:

```
CustomerSatisfaction >> title
  ^ 'Customer Satisfaction Survey'.

CustomerSatisfaction >> extent
  ^ 400@100
```

The `title` method returns a string that will be used as a title, and the `extent` method returns a point that specifies the size of the window.

To open a UI, an instance of the class needs to be created and it needs to be sent the `openWithSpec` message. This will open a window and return an instance of `WindowModel`, which allows the window to be closed from code.

```
| ui |
ui := CustomerSatisfaction new openWithSpec.
[ ... do a lot of stuff until the UI needs to be closed ...]
ui close.
```

More information about managing windows: e.g., opening dialog boxes or setting the about text is present in Chapter 6.

This concludes our first example of a Spec user interface. We now continue with more examples on how to configure the different widgets that can be used in such a user interface.

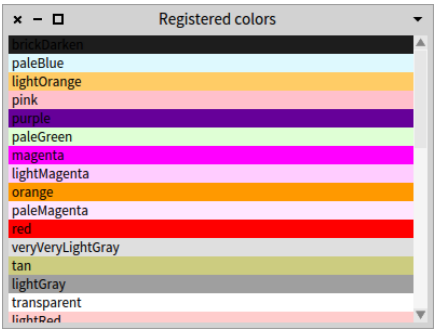

Figure 2-2 Screen shot of the list with modified background colors

2.2 Fun with Lists

As an illustration of how widgets can be configured, we now show two examples of lists: a list using different background colors and a list that also shows icons.

These examples show two important features of Spec:

1. All widgets can be opened as a window. This is because there is no fundamental difference between a complex UI as shown above and the standard widgets.
2. All widgets can be configured, and their configuration methods are classified in `api` protocols.

Registered colors as item background

We start with an example of a `ListModel` where the elements have different background colors, shown in Figure 2-2. The items held in the list are the names of different colors, and the list shows them using a background of that color.

The following code shows how this is done:

```
| registeredColorsList |
registeredColorsList := ListModel new.
registeredColorsList
 items: Color registeredColorNames;
 backgroundColorBlock: [ :item | Color named: item ];
 title: 'Registered colors'.
registeredColorsList openWithSpec
```

Here we see the following messages that are part of the `ListModel` API.

- The message `items:` sets the elements of the list.

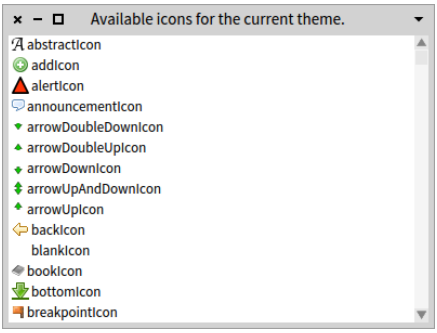

Figure 2-3 Screen shot of the list of Icons

- The message `backgroundColorBlock:` specifies a block that is executed to determine the background color of the current element. It takes a block with a single parameter: the list item.
- The message `title:` sets the title of the window containing the list.

List of icons

The second example shows a list containing the icons of the current theme and their respective selector in an `IconListModel`. The list items are associations of the icon names and the icons themselves, the text that is shown in the list is the icon name, and the icon shown in the list is the icon itself. We also sort the items in the list alphabetically according to their name.

```
| iconList |
iconList := IconListModel new.
iconList
 items: Smalltalk ui icons icons associations;
 displayBlock: [ :assoc | assoc key];
 sortingBlock: [ :assocA :assocB | assocA key < assocB key ];
 icons: [ :assoc | assoc value ];
 title: 'Availiable icons for the current theme.'.
iconList openWithSpec
```

The following messages of the `ListModel` API are noteworthy:

- The message `displayBlock:` takes a block that receives a domain specific item and should return something that can be displayed in a list, like a String.
- The message `sortingBlock:` takes a block that is used to sort the elements of the list before displaying them.

2.3 Conclusion

In this chapter we have given you a first contact with Spec user interfaces. We have first shown you what the different steps are to build a user interface with Spec, and then shown you two examples of how to configure existing Spec widgets.

More examples of Spec user interfaces are found in the Pharo Image itself. Since all Spec user interfaces are subclasses of `ComposableModel`, they are easy to find and each of them may serve as an example. Furthermore, experimentation with widgets and user interfaces is made easy because all widgets can be opened as standalone windows, and their configuration methods are classified in the `api` protocol.

We recommend that you at least read the next chapter about reuse of Spec widgets, which is the key reason behind the power of Spec. This knowledge will help you in building UIs faster through better reuse, and also allow your own UIs to be reused. The chapter after that on the three pillars of Spec gives a more complete overview of the functioning of Spec and is worthwhile to read in its entirety. Later chapters are intended more as reference material for specific problems or use cases, but can of course be read in full as well.

CHAPTER 3

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.

3.1 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 3-1 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.

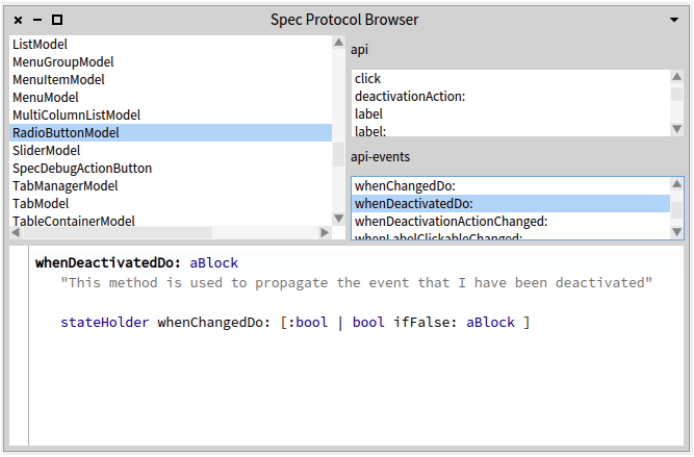

Figure 3-1 Screen shot of the ProtocolBrowser.

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.

3.2 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.
```

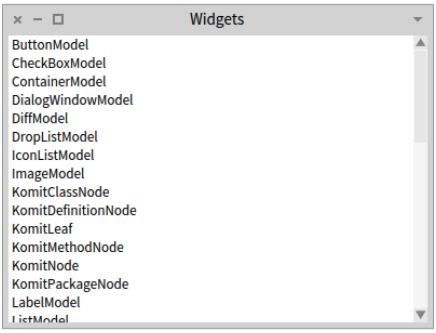

Figure 3-2 Screen shot of the WidgetClassList.

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

The layout contains only the list:

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

The resulting UI is shown in Figure 3-2.

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 `whenSelectedItemChanged:`, 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 public 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. We will discuss the topic of a public configuration API in Section 3.6 at the end of this chapter.

3.3 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 5 for more on layouts.)

```
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 3-3 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:

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

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

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

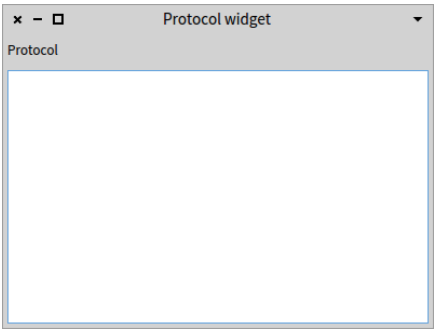

Figure 3-3 Screen shot of the ProtocolMethodList.

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

> **Note** An alternative to adding these methods is simply to do nothing: since both the methods and the label are accessible (through their accessors), a reuser of this widget may simply obtain them and configure them directly. These two alternatives reflect a design decision that we will discuss in Section 3.6.

3.4 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 3-4. This user interface is functional, clicking on a class will show the methods of the `api` and the `api-events` protocols of that class.

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.

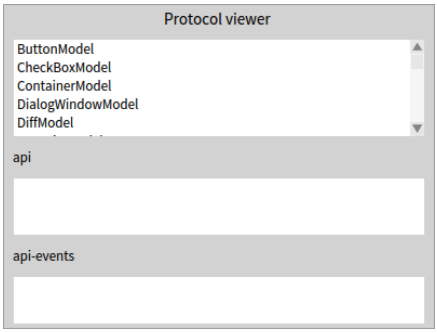

Figure 3-4 Render of the ProtocolViewer.

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

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

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

> **Note** Compared to the public configuration API methods we have seen before, these methods add semantic information to the configuration API. They state that they configure what to do when a class, `api` or `api-events` list item has been changed. This arguably communicates the customization API more clearly than just having the subwidgets accessible.

3.5 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
      ifNotNil: [ text text: item sourceCode ] ].
  viewer whenEventChanged: [ :item |
    item
      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 3-1.

3.6 Considerations about a public configuration API

In this chapter we have seen several definitions of methods in the public configuration API of the widget being built. The implementation of our configuration methods here simply delegated to internal widgets, but a configuration can of course be more complex than that, depending on the internal logic of the UI.

For methods that simply delegate to the internal widgets, the question is whether it makes sense to define these as methods in the `api` protocols at all. This fundamentally is a design decision to be made by the programmer. Not having such methods makes the implementation of the widget more lightweight but comes at a cost of a less clear intent and of breaking encapsulation.

For the former cost, we have seen an example in the protocol method list of Section 3.3. The presence of the three methods defined there communicates to the user that we care about what to do when a class, `api` or `api-events` list item has been changed. The same fundamentally also holds for the other examples in this chapter: each method in an `api` protocol communicates an intent to the reuser: this is how we expect that this widget will be configured. Without such declared methods, it is less clear to the reuser what can to be done to be able to effectively reuse this widget.

For the latter cost, expecting reusers of the widget to directly send messages to internal objects (in instance variables) means breaking encapsulation. As a consequence, we are no longer free to change the internals of the UI, e.g., by renaming the instance variables to a better name or changing the kind of widget used. Such changes may break reusers of the widget and hence severely limits how we can evolve this widget in the future. In the end, it is safer to define a public API and ensure in future versions of the widget that the functionality of this API remains the same.

So in the end it is important to consider future reusers of your UI and future evolution of your UI. You need to make a tradeoff of writing extra methods versus possibly making reuse of the UI harder as well as possibly making future evolution of the UI harder.

3.7 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 public customization methods should be placed in a `api` protocol.

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.

CHAPTER 4

The fundamentals of Spec

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

4.1 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. The model that is defined in Spec corresponds to a presenter in the MVP triad. Note that because of this, future versions of Spec may rename the `ComposableModel` into `ComposablePresenter` for naming consistency.

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.

4.2 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.`

4.3 Defining UI Layouts

Widget layout is defined by specifying methods that state how the different widgets that compose a UI are placed. In addition, it also specifies how a widget reacts when the window is resized. As we will see later, these methods can have different names and are subject to a specific lookup process.

A 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 a layout method is mandatory, as without it the UI would show no widgets to the user.

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.

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: ''
```

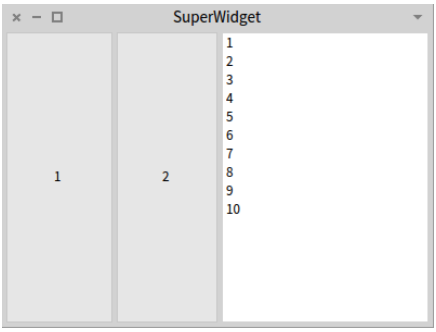

Figure 4-1 Screen shot of the UI with buttons placed horizontally

```
  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 4-1. 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 reused `buttons` widget should use the `buttonCol` layout method, and hence results in the window shown in Figure 4-2.

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

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

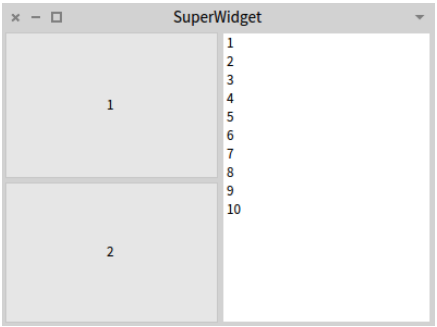

Figure 4-2 Screen shot of the UI with buttons placed vertically

4.4 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 primitive methods, the basic widgets provide more specific hooks, e.g., when an item in a list is selected.

4.5 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. 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.

CHAPTER 5

Layout Construction

Placing and describing the behavior of widgets on resizing of their container is an important yet complex issue. In this chapter we present the different ways to express layouts with Spec.

5.1 About layouts

Layout of widgets in a window or in their reusing UI is a nontrivial problem. This is because there are many factors at play in determining where a widget should be placed in its container (the container can be a window or another widget).

A straightforward solution is to place widgets at absolute coordinates of the enclosing container, but this breaks when the container is resized: widgets do not grow or shrink together with the container. However, if the window is never resized this is not really an issue either, and absolute positioning allows for pixel-perfect placing of every widget.

Since there are multiple usage scenarios, multiple ways for widgets to be laid out need to be provided. Spec provides various options to perform layouts and these are visible as methods in the `SpecLayout` class, in the `commands` protocols. Up until now, as layouts go we have only seen the use of rows and columns, and the adding of a single widget to the container. Furthermore, in all of these results, a row, column or a single widget always took up all available space in its container, which is the default behavior.

The `add:` method on `SpecLayout` only allows one widget to be added, so if you want more than one widget in your user interface you will need to specify one of the layouts we present in this chapter. We discuss in depth two broad strategies for laying out widgets: first the various options for speci-

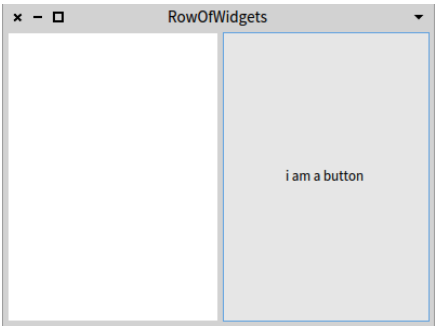

Figure 5-1 Screen shot of a row of widgets.

fying rows and columns, and second the diverse ways in which widgets can be laid out freely. Each of them has their advantages and disadvantages depending on the kind of UI that is being constructed so it is best to know all of them well enough to be able to make the tradeoff.

A working example

To illustrate these layouts, we use an example class that has two buttons, a list and a text field, the non-layout code of which is below:

```
ComposableModel subclass: #LayoutExample
  instanceVariableNames: 'list button button2 text'
  classVariableNames: ''
  package: 'Spec-BuildUIWithSpec'
```

```
LayoutExample >> initializeWidgets
  button := self newButton.
  button2 := self newButton.
  list := self newList.
  text := self newText.
  button label: 'i am a button'.
  button label: 'me too!'.
```

5.2 Row and column layouts

A straightforward and often used kind of layout to have more than one widget is to have the widgets aligned in rows or columns. Spec provides for an easy way to specify such layouts through the use of the `newRow:` and `newColumn:` messages on the `SpecLayout` class. They create, respectively, a row or a column in which the space given to the widgets is evenly distributed.

For example, the code below lays out the list and the button in a row:

Figure 5-2 Screen shot of a column of widgets.

```
LayoutExample >> oneRow
  ^ SpecLayout composed
    newRow: [ :row | row add: #list; add: #button ];
    yourself
```

This code is a layout method that builds a row of widgets using the `newRow:` message. The argument of the message is a one-argument block, and the block argument will contain an instance of a `SpecRowLayout`. The widgets are then added to this layout object, which aligns them all in a row.

The code below opens the example with this layout specification, resulting in the UI shown in Figure 5-1. The `title:` message is self-explanatory.

```
| le |
 le := LayoutExample new.
 le title: 'RowOfWidgets'.
 le openWithSpec: #oneRow
```

Having the widgets rendered as a column is similar, the only real difference being that instead of the `newRow:` message, a `newColumn` message is used. This message also takes a one-argument block, and the block argument will contain an instance of a `SpecColumnLayout`.

```
LayoutExample >> oneColumn
  ^ SpecLayout composed
    newColumn: [ :col | col add: #list; add: #button ];
    yourself
```

The code below opens this example, resulting in Figure 5-2. Note that for brevity, in the remainder of this chapter we will not include any more UI opening code as it is straightforward.

```
| le |
 le := LayoutExample new.
 le title: 'ColumnOfWidgets'.
 le openWithSpec: #oneColumn.
```

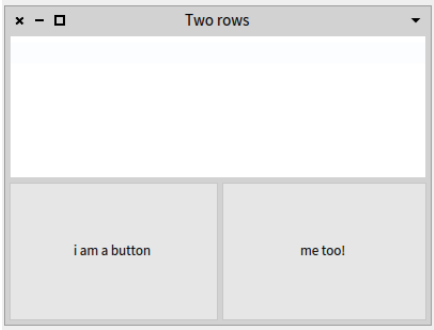

Figure 5-3 Screen shot of two rows.

Spec also allows for using the `SpecRowLayout` and `SpecColumnLayout` directly instead of using a `SpecLayout`. This makes the above code for the `oneRow` and `oneColumn` layout methods more concise:

```
LayoutExample >> oneRowConcise
  ^ SpecRowLayout composed
      add: #list; add: #button;
      yourself.
```

```
LayoutExample >> oneColumnConcise
  ^ SpecColumnLayout composed
      add: #list; add: #button;
      yourself
```

5.3 Combining rows and columns

Rows and columns can be combined to build more complex layouts, by sending both the `newRow:` and `newColumn:` messages in different combinations. We show some examples here to illustrate the possibilities.

The first example shows how we can have two rows of widgets: we create a column and twice call `newRow:`. The result is shown in Figure 5-3.

```
LayoutExample >> twoRows
  ^ SpecColumnLayout composed
      newRow: [ :row | row add: #text ];
      newRow: [ :row | row add: #button; add: #button2 ];
      yourself
```

> **Note** To have multiple rows, these need to be added to a `SpecColumnLayout`, and to have multiple columns, these need to be added to a `SpecRowLayout`. Sending `addRow:` or `addColumn:` to a `SpecComposedLayout` multiple times will only produce the last row, resp. column.

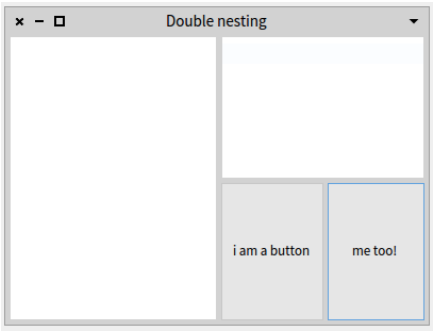

Figure 5-4 Screen shot of multiply-nested rows and columns.

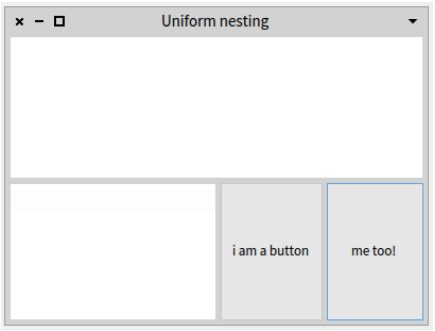

Figure 5-5 Screen shot of multiply-nested rows.

Rows and columns can of course also be multiply-nested, for example here we add the two buttons in a row that is nested in a column that is nested in a `SpecRowLayout`. The resulting UI is shown in Figure 5-4.

```
LayoutExample >> nesting1
  ^ SpecRowLayout composed
    newColumn: [ :col | col add: #list];
    newColumn: [ :col |
      col
        add: #text;
        newRow: [ :row |
          row
            add: #button;
            add: #button2]
    ];
    yourself
```

In addition to nesting columns in rows (and vice-versa), rows can also be nested in rows (and columns in columns), which allows the used space per

widget to be uniformly halved, as shown in Figure 5-5

```
LayoutExample >> nesting2
  ^ SpecColumnLayout composed
    newRow: [ :row | row add: #list];
    newRow: [ :row |
      row
        add: #text;
        newRow: [ :inRow |
          inRow
            add: #button;
            add: #button2]
    ];
    yourself
```

5.4 Setting row and column size

By default, rows and columns take up all available space, and the space in a row (and in a column) is evenly distributed across all elements of that row (or column). In this section we show three different ways in which the size of rows and columns can be changed. The first is by letting the user resize them and the two last are two different ways in which their size can be specified.

Adding UI splitters so the user can resize

A simple resizing option is to allow the user to resize widgets horizontally (for rows) and vertically (for columns). This is done by adding an `addSplitter` message between the `add:` messages for the widgets of that row or column. For example, the code below makes the horizontal line between the list and the button draggable up or down. Visually the result is the same as in Figure 5-2.

```
LayoutExample >> oneColumnWithSplitter
  ^ SpecColumnLayout composed
      add: #list;
      addSplitter;
      add: #button;
      yourself
```

Size in pixels

Programmatically it is possible to explicitly specify the absolute size of a row or column in pixels by using the `newRow:height:` and `newColumn:width:` methods. This is useful, for example, if a row of buttons is to be placed above or below a text field where it can avoid an ugly layout with huge buttons as seen previously in Figure 5-3.

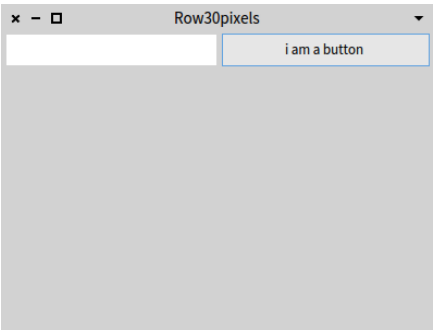

Figure 5-6 Screen shot of a row of 30 pixels high.

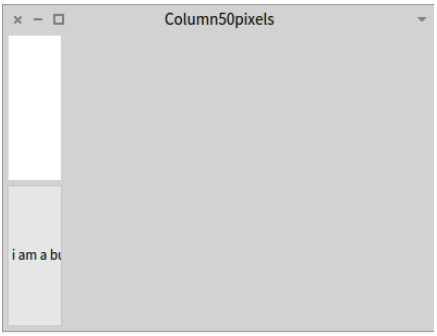

Figure 5-7 Screen shot of a column of 50 pixels wide.

We show two simple examples of the use of absolute size below, the first leading to the UI of Figure 5-6 and the second to the UI of Figure 5-7.

```
LayoutExample >> rowOf30
  ^ SpecLayout composed
    newRow: [ :row | row add: #list; add: #button] height: 30;
    yourself
```

```
LayoutExample >> columnOf50
  ^ SpecLayout composed
    newColumn: [ :col | col add: #list; add: #button] width: 50;
    yourself
```

Alternatively, inside of a column layout and a row layout, `add:height:`, respectively `add:width:`, can also be used to state the height, respectively the width, of that specific widget.

> **Note** It is considered a bad practice to hardcode the size of widgets in pixels, as some changes (e.g., in font size) can invalidate this number. To

> alleviate this, `ComposableModel class` provides accessors for practical sizes in its `defaults` protocol.

An example of the use of accessors that compute height is `toolbarHeight`, which we used in the protocol method browser, shown in Figure 3-3. This accessor depends on the font size and is also useful for sizing button rows.

Proportional layout

A last option is to specify the percentage of the container, e.g., the window, that a row or column should occupy. This is performed using the messages `newRow:top:bottom:` and `newColumn:left:right:`. In contrast with specifying size in pixels, the use of these messages will cause the row or column size to change accordingly when the container is resized.

Both the above messages take two numbers as extra arguments. These should be between 0 and 1: they are a percentage that states how far **towards the other edge** the element should begin, resp. end. For example, a column that starts at the left end of the window and takes 30 percent of the width is a `newColumn: [:c| ...] left: 0 right: 0.7` because 30 percent of the width is 70 percent away from the right edge.

> **Note** Both these numbers indicate a percentage that is towards 'the other end' of the container, **not** a percentage from top to bottom or from left to right.

A more complex example is the code below, an arguably artificial variant on Figure 5-5 that makes the buttons proportionally smaller. (The example is artificial because, e.g., for the row height of the buttons it would make more sense to use sizes in pixels.) It states that the top row takes the first 80 percent of the space (since `top: 0 bottom: 0.2`) and the second row the last 20 percent (since `top: 0.8 bottom: 0`). Furthermore, in the bottom row, the text field takes up the first 55 percent of the space (`left: 0 right: 0.45`) and the two buttons the last 45 percent (`left: 0.55 right:0`). The result of this code can be seen in Figure 5-8.

```
nestingTB
  ^ SpecColumnLayout composed
    newRow: [ :row | row add: #list] top: 0 bottom: 0.2 ;
    newRow: [ :row |
      row
        newColumn: [:c | c add: #text] left: 0 right: 0.45;
        newColumn: [ :c |
          c newRow: [ :inRow |
            inRow
              add: #button;
              add: #button2]] left: 0.55 right: 0
    ] top: 0.8 bottom: 0 ;
```

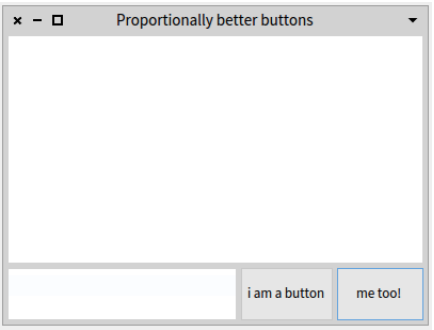

Figure 5-8 Screen shot of a use of proportional rows and columns.

```
    yourself
```

5.5 Layouts without rows or columns

Rows and columns are of course not the only way in which a user interface can be laid out. Spec also allows widgets to be placed more freely. This can be done either using absolute positions of the enclosing container or using relative positioning, which takes into account window resizing. We show here how these different placement options can be used.

Absolute widget positions

A first manner in which widgets can be laid out, is by giving them absolute positions in their enclosing container. To do this, the `SpecLayout` method `add:top:bottom:left:right:` is used. It takes four extra arguments, each representing a distance as a number of pixels. The number of pixels should be positive, as it indicates a distance from the given edge towards the opposite edge.

For example, below we perform a layout of a button at 10 pixels from the top, 200 pixels from the bottom, 10 from the left and 10 from the right, and the result is shown in Figure 5-9.

```
LayoutExample >> oneButtonAbsolute
  ^ SpecLayout composed
    add: #button top: 10 bottom: 200 left: 10 right: 10;
    yourself
```

> **Note** The underlying logic of the arguments is the same as in `newRow:top:bottom:` and `newColumn:left:right` of Section 5.4: distances are towards the 'other end' of the container.

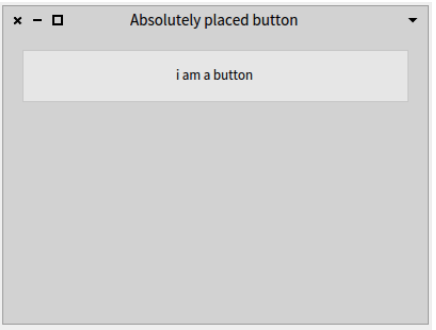

Figure 5-9 Screen shot of an absolutely placed button

> **Note** Obviously, the use of absolute widget positions breaks completely if a window is resized. It is therefore best to use this only in combination with windows that cannot be resized.

Relative widget positions

Relative widget positions cause the widget to be resized according to how its container is resized. The method `add: origin: corner:` of `SpecLayout` specifies such relative layout of a widget, percentage-wise from the origin point to the corner point. These two points represent respectively the top left corner and the bottom right corner of the widget. Since the arguments express a percentage of the container, they must be between 0@0 and 1@1.

For example, below we place one button that is half the container size in the center of the container, which can be seen in Figure 5-10.

```
LayoutExample >> oneButtonSmaller
  ^ SpecLayout composed
    add: #button origin: (0.25 @ 0.25) corner: (0.75 @ 0.75);
    yourself
```

Alternatively, the method `add:top:bottom:left:right:` can also be used with percentage arguments, i.e., between 0 and 1, to produce a relative layout with percentages computed from the opposite edge. For example, the code below produces exactly the same result as the code above. We however discourage this use of the `add:top:bottom:left:right:` method for two reasons. Firstly there may be confusion between relative and absolute: should you interpret a value of 0 or 1 as a percentage or as a number of pixels? Secondly it does not support the use of offsets, which we discuss next.

```
LayoutExample >> oneButtonSmallerAlternative
  ^ SpecLayout composed
    add: #button top: 0.25 bottom: 0.25 left: 0.25 right: 0.25;
    yourself
```

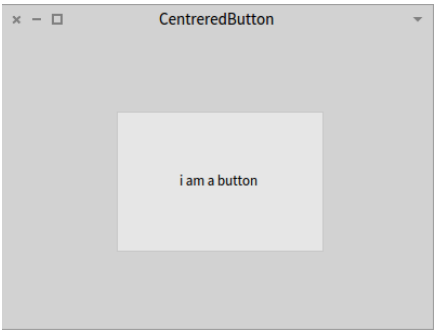

Figure 5-10 Screen shot of an always centered button

Relative with offsets

Widget relative positions have the advantage that they resize widgets as their container grows and shrinks, but have the disadvantage that widgets are always placed right next to each other or next to the container border. Placing a widget at, e.g., five percent of a container border, can make the border of the correct thickness for a specific window size but then too big or too small when the window is resized. Actually, what is needed is for a way in which we can place widgets relatively, but specify absolute offsets in addition, so that the gaps between widgets will always be the same. This is possible with the `add: origin: corner: offsetOrigin: offsetCorner:` method of `SpecLayout`. The `origin: corner:` arguments are the same as in the `add: origin: corner:`method and both offset arguments take a point as argument. These points express the number of pixels from the corresponding corner, in the classical computer graphics coordinate system where the origin is in the top left corner. Consequently, the `x` or `y` component can be negative representing an offset towards the left, respectively the top.

For example, the code below lays out two buttons on top of each other. Each of them takes half the space of the window, minus the window border of 10 pixels and a the space between them of 10 pixels.

```
LayoutExample >> twoButtonsRelativeOffset
  ^ SpecLayout composed
    add: #button origin: (0 @ 0) corner: (1 @ 0.5)
         offsetOrigin: (10 @ 10) offsetCorner: (-10 @ -5);
    add: #button2 origin: (0 @ 0.5 ) corner: (1 @ 1)
         offsetOrigin: (10 @ 5) offsetCorner: (-10 @ -10);
    yourself
```

Relative widget positions for rows and columns

Lastly, for easy composition of rows and columns with other widgets, both rows and columns can be placed relatively as well as relatively with an offset. The methods `newRow:` and `newColumn:` have their variants with suffix `origin: corner:` and `origin: corner: offsetOrigin: offsetCorner:`. These add the relative layout options we have seen for widgets to rows and columns.

One example of the use of this placement was in the very first chapter, in the customer satisfaction example. The layout code for this example is below, and the result can be seen in Figure 2-1.

```
CustomerSatisfaction class >> defaultSpec
  ^ SpecLayout composed
    newRow: [ :row |
        row add: #buttonHappy; add: #buttonNeutral; add: #buttonBad ]
      origin: 0 @ 0 corner: 1 @ 0.7;
    newRow: [ :row |
        row add: #screen ]
      origin: 0 @ 0.7 corner: 1 @ 1;
    yourself
```

5.6 Conclusion

In this chapter we have discussed the various layout strategies that Spec provides. If your user interface has more than one widget, you will need to use one of these strategies. We talked about two broad strategies of layouts: firstly row and column based layouts and secondly absolute and relative layouts (and their combination). It is prudent to know the advantages and disadvantages of each layout when constructing your UI so the choice of layout that is used corresponds to a good tradeoff considering the visual appearance of the user interface.

CHAPTER 6

Managing windows

In this book so far we have talked about reuse of `ComposableModels`, discussed the fundamental functioning of Spec and presented how to layout the widgets of a user interface. Yet what is still missing for a working user interface is showing all these widgets inside of a window. In our examples until now we have only shown a few of the features of Spec for managing windows, basically restricting ourself to opening a window.

In this chapter we provide a more complete overview of how Spec allows for the managing of windows. We show opening and closing, the built-in dialog box facility, sizing of windows and all kinds of window decoration.

A working example

To illustrate the window configuration options that are available, we use a simple `WindowExample` class that has two buttons placed side by side. These buttons do not have any behavior associated yet, this will be added in an example further down this chapter.

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

```
WindowExample >> initializeWidgets
   button1 := self newButton.
   button2 := self newButton.
   button1 label: '+'.  button2 label: '-'.
```

```
WindowExample class >>defaultSpec
   ^ SpecRowLayout composed
         add: #button1; add: #button2; yourself.
```

6.1 Opening a window or a dialog box

A user interface can be opened as a normal window, or opened as a dialog box, i.e. without decoration and with *Ok* and *Cancel* buttons. We show here how this is done, including the configuration options specific to dialog boxes. (See also Section 6.3 for more information about window decoration.)

Opening a window

As we have shown in previous sections, to open a user interface you need to instantiate the `ComposableModel` for that interface and send it the `openWithSpec` message. This creates an instance of `WindowModel` containing the user interface and shows it in a window on screen.

We have also seen the `openWithSpec:` method, notably in Chapter 5, that takes the name of a layout message as argument. Instead of using the default layout (whose lookup is described in Section 4.3), the opened UI will use the layout returned by that method. A second variant is `openWithSpecLayout:` that takes a `SpecLayout` instance (or an instance of its subclasses).

For example, below we show the three ways we can open a window for our `WindowExample`. It will open three identical windows.

```
| we |
we := WindowExample new.
we openWithSpec.
we openWithSpec: #defaultSpec.
we openWithSpecLayout: we defaultSpec.
```

Opening a dialog box and its configuration options

Spec provides for an easy way to open a UI as a simple dialog box with *Ok* and *Cancel* buttons (that has no icons for resizing, closing or the window menu). To do this, send the message `openDialogWithSpec` as below:

```
| we diag |
we := WindowExample new.
diag := we openDialogWithSpec.
```

The result of this (e.g. assigned to the `diag` variable above) is an instance of the `DialogWindowModel` class (a subclass of `WindowModel`). For convenience, `openDialogWithSpec` has the same variants as `openWindowWithSpec` that take a specific layout as argument, either as a message name or as a `SpecLayout` instance.

The `DialogWindowModel` instance (`diag` in the example above) can also be configured in multiple ways. To execute code when the user clicks on a button, send it the `okAction:` or `cancelAction:` message with a zero-argument block. If the block returns `false` the window will not be closed

(if it returns anything else it will be closed). This closing logic allows for the validation of the contents of the dialog box, only allowing it to be closed if validation succeeds. Complimentarily, the *OK* button can be grayed out by sending the `okButtonEnabled: false` message, and enabled again with a `true` argument.

Also, the message `cancelled` will return `true` if the box was closed by clicking on the *Cancel* button.

Taking over the entire screen

Lastly, it is also possible to open a UI such that it takes over the entire Pharo screen. There are no other windows, the UI is full screen without any resizing or closing boxes, nor a title bar. The Pharo window is your UI's window.

To take over the entire Pharo screen, send the message `openWorldWithSpec`:

```
WindowExample new openWorldWithSpec.
```

> **Note** Via the halos the UI can still be closed, but any other windows that were open before will have disappeared.

6.2 Modal windows and the closing of windows

Windows are not alone on the screen, and they probably will not live forever. Here we talk about gaining full control of the entire user interface, and how to hook into the window closing logic.

Modal windows

A modal window is a window that takes control of the entire Pharo user interface, making it impossible for the user to select another window while it is open. This is especially useful for dialog boxes, but may also be needed for other kinds of windows.

Spec allows any window to be made modal by sending the message `modalRelativeTo: World` to the `WindowModel` that corresponds to the opened `ComposableModel`. To get a handle on the `WidowModel`, send the `window` message (after the UI has been opened with `openWithSpec`).

```
| we |
 we := WindowExample new.
 we openWithSpec.
 we window modalRelativeTo: World
```

> **Note** The argument of `modalRelativeTo:` should always be the root of the currently shown widget hierarchy: `World`.

Preventing window close

Spec provides for the possibility to check if a window can effectively be closed when the user clicks on the close box. To use it, this feature must first be turned on, by sending `askOkToClose: true` to the `ComposableModel`. This can be done for example by changing our `WindowExample` as follows:

```
WindowExample >> initializeWidgets
   button1 := self newButton.
   button2 := self newButton.
   button1 label: '+'.
   button2 label: '-'.
   self askOkToClose: true.
```

The behavior of the close button however is still not changed, closing a window is still possible. This is because we have not defined the implementation of what to check on window close. This is most easily done by overriding the `okToChange` method of `ComposableModel`, for example as below:

```
WindowExample >> okToChange
   ^false
```

Because this method returns `false`, clicking on the close button of an open `WindowExample` window will not have any effect. We have effectively created an unclosable window! To be able to close this window, we should change the implementation of the above method to return `true` (or simply remove it).

Of course the example `okToChange` method above is extremely simplistic and not very useful. It instead should define application-dependent logic of what to check on window close. Note that there are many examples of `okToChange` methods in the system that can be used as inspiration.

Acting on window close

It is also possible to perform an action whenever a window is closed, by sending the `whenClosedDo:` message to the UI's `WindowModel`. For example, below we specify a goodbye message to deliver when our example UI is closed.

```
| we |
 we := WindowExample new.
 we openWithSpec.
 we window whenClosedDo: [ UIManager default inform: 'Bye bye!' ].
```

6.3 Window size and decoration

We now focus on sizing a window before and after opening it, and then talk about removing the different control widgets that decorate the window.

Setting initial size and changing size

To set the initial size of a window when it opens, either override the `extent` method of the corresponding `ComposableModel` so that it returns a Point, or send the instance the `extent:` message before opening, for example like this:

```
| we |
 we := WindowExample new.
 we extent: 300@80.
 we openWithSpec
```

After a window is opened, it can also be resized by sending the `extent:` message to the window of the UI. For example, we can change our examples' `initializeWidgets` method so that the window resizes itself depending on what button is clicked.

```
WindowExample >> initializeWidgets
  button1 := self newButton.
  button2 := self newButton.
  button1 label: '+'.
  button2 label: '-'.
  button1 action: [ self window extent: 500@200].
  button2 action: [ self window extent: 200@100].
```

Fixed size

The size of a window can be made fixed, so that the user cannot resize it by dragging the sides or corners. To configure this we however need to talk to the underlying widget library (`Morphic` in Pharo 5). We get the `Morphic` window of our example (via its `WindowModel`) and instruct it to be unresizable as follows:

```
| wewin |
 wewin := WindowExample new openWithSpec.
 wewin window beUnresizeable
```

Removing window decoration

Sometimes it makes sense to have a window without decoration, i.e. without control widgets. Currently this configuration cannot be performed on the `ComposableModel` of that window, but the underlying widget library may allow it. Below we show how to get the `Morphic` window of our example and instruct it to remove the different control widgets:

```
| wewin |
 wewin := WindowExample new openWithSpec.
 wewin window
    removeCollapseBox;
    removeExpandBox;
```

```
    removeCloseBox;
    removeMenuBox
```

Note This window is still closable using the halo menus or by calling `close` on the `WindowModel` instance (`wewin` in the example above).

6.4 The final details: title, icon and about text

You can provide some textual information about the widow by providing a title and a text for the window's about dialog, as we show here.

Setting and changing the title

By default, the title of a new window is *'Untitled window'*. This can be changed by overriding the `title` method (of `ComposableModel`) and returning a string that will be used as a title. For example, we can title our example user interface as follows:

```
WindowExample >> title
   ^ 'Click to grow or shrink.'
```

In addition, you can set the title of any UI after it has been opened (even if it specifies a `title` method) by sending the `title:` message with the new title as argument to the window of the UI. An example is below:

```
| we |
 we := WindowExample new.
 we openWithSpec.
 we window title: 'I am different!'
```

Setting the icon

At the bottom of the main Pharo window there is a window taskbar, allowing the user to switch between windows by clicking on the buttons that represent each window. These buttons also have an icon that is ment to represent the windows' kind. This icon can also be configured through Spec, in two different ways.

Firstly, sending the `windowIcon:` message to the `ComposableModel` allows an icon to be set per window, as below. Note that it does not matter if the message is sent before or after the window is opened.

```
| wm1 wm2 |
 wm1 := WindowExample new.
 wm1 openWithSpec.
 wm1 windowIcon: (Smalltalk ui icons iconNamed: #thumbsDown).
 wm2 := WindowExample new.
 wm2 windowIcon: (Smalltalk ui icons iconNamed: #thumbsUp).
 wm2 openWithSpec.
```

Secondly, the icon can be changed by overriding the `windowIcon` message, as below. (The code below is for Pharo 6, for this to work in Pharo 5 replace `self iconNamed:` with `Smalltalk ui icons iconNamed:` .)

```
WindowExample >> windowIcon
   ^ self iconNamed: #thumbsUp
```

> **Note** Changing the `windowIcon` method will affect all open windows, as the taskbar is periodically refreshed. This refreshing is also why `windowIcon:` can be sent before or after the window has been opened.

Setting the about text

To set the about text of a window, either override the `aboutText` method of the corresponding `ComposableModel` so that it returns the new about text, or send the instance the `aboutText:` message before opening, for example like below.

```
| we |
 we := WindowExample new.
 we aboutText: 'Click + to grow, - to shrink.'.
 we openWithSpec
```

6.5 Conclusion

In this chapter we treated the features of Spec that have to do with windows. We first talked about opening and closing windows as well as how to open a window as a dialog box. This was followed by configuring the window size and its decorating widgets. We ended this chapter with the small yet important details of the window: its title, icon and about text.

CHAPTER 7

Advanced Widgets

Some Spec user interface elements provide more advanced functionalities and therefore are more complex to configure than, e.g. a button or a label. In this chapter we show four such advanced widgets and present how they can be used and configured. We first discuss text input widgets, then treat radio buttons and tabs, and finish with toolbars and pop-up menus.

7.1 TextModel

In Spec, the input of text by the user or the showing of multiline text is done using a `TextModel`. We now discuss some typical configurations of this model, as well as the extras provided by the one-line `TextInputFieldModel`.

Non-editable text field

The text field of `TextModel` is editable by default, but sometimes it is useful for a UI to show multiple lines of text that are non-editable. To achieve this, simply send `disable` to the model, for example as follows.

```
| cm |
cm := TextModel new.
cm text: Object comment.
cm disable.
cm openWithSpec.
```

The above example shows the class comment of the class `Object`, in a non-editable way. Note that the text still can be selected, copied and searched.

Remove the yellow triangle, perform an action at each edit

`TextModel` provides visual feedback when the text has been edited by placing a yellow triangle in the top right of the field. Also, if the widget is removed, e.g. by closing the window it contains, a confirmation dialog will be presented to the user. This 'has been edited' flag is reset by calling the `accept` message on the model. Sending the message `accept` means telling the widget to make sure that the changed text is not lost if the widget is closed, which is why the user is not notified about a possible loss.

For an example, execute the code below, and when the UI opens type a few characters. After 3 seconds the triangle will disappear and closing the window does not require confirmation of the user.

```
| cm |
cm := TextModel new.
cm openWithSpec.
[3 seconds wait. cm accept] fork
```

The model can also be configured to automatically accept at each keystroke, effectively removing the functionality of the edit flag. This is done by sending it `autoAccept: true`.

What actually happens in the `accept` call is configurable. The default action is to set the `text` field of the model to the text held in the user interface. To add to this behaviour, send the model the `acceptBlock:` message, which takes a one-argument block. The argument of the block is the text held in the user interface.

The following example shows how to combine the `autoAccept` and `acceptBlock:` messages to create a text field that shows its text as a growl morph whenever it is edited.

```
| cm |
cm := TextModel new.
cm autoAccept: true.
cm acceptBlock: [ :txt | GrowlMorph openWithContents: txt.].
cm openWithSpec.
```

Keyboard shortcuts

With text entry fields, there is sometimes also a need to provide keyboard shortcuts for actions that go beyond the usal copy-and-paste actions. Defining such custom actions is done using the `bindKeyCombination:action:` message.

> **Note** A keyboard shortcut does not need to be a key in combination with a modifier like command. For example it can be simply the letter `x`, or the space character.

For example, the code below opens an inspector on the text when command-i is pressed. Note that we need to set auto-accept to `true` for `text` to always hold the characters that were entered in the widget.

```
| cm |
cm := TextModel new.
cm autoAccept: true.
cm bindKeyCombination: $i command toAction: [ cm text inspect ].
cm openWithSpec
```

The `bindKeyCombination:action:` method is actually defined in `ComposableModel` and can therefore be used on any UI widget. It is used in different standard widgets to bind keyboard actions that are not necessarily keyboard shortcuts. For example, the last line of the initializer of `ButtonModel` binds the execution of the button's action to the space bar, as follows:

```
self bindKeyCombination: Character space toAction: [ self action ].
```

Single-line input field extras

`TextInputFieldModel` is a subclass of `TextModel` that is meant for single-line inputs. Pressing enter or return does not cause a carriage return, instead the accept action of the field is executed. This widget is however still able to display multiple lines of text, for example if multi-line text is pasted in. But it will not show a scrollbar if it runs out of vertical space for the text.

It also adds a few features that are useful:

- A ghost text can be set with the `ghostText:` message.
- Password fields that show only stars are specified by sending the `beEncrypted` message.
- Entry completion can be added by using the `entryCompletion:` message. The argument must be an `EntryCompletion` instance, and we refer to that class for examples.
- Sending `acceptOnCR: false` to the input field inhibits the accept action to be executed when enter or return is pressed, and as a consequence the user can enter multiple lines of text.

7.2 RadioButtonModel

Radio buttons allow the user to select at most one option from a group, and unlike dropdown menus all items of this group are visible on screen. The `RadioButtonModel` of Spec uses `RadioButtonGroup` to manage the group.

As an example UI we present a basic washing machine control panel, shown in Figure 7-1. There are two groups of radio buttons: one for the kind of fab-

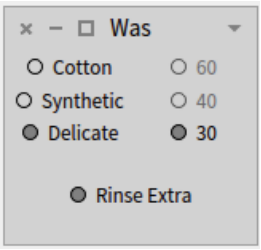

Figure 7-1 Screen shot of the washing machine control panel.

ric and one for the temperature of the water (in Celcius). Lastly, there is one more radio button that allows for an extra rinse cycle to be selected.

The code for the UI is in a `RadioButtonExample` class, whose definition and layout are straightforward:

```
ComposableModel subclass: #RadioButtonExample
  instanceVariableNames: 'rinse f1 f2 f3 t1 t2 t3'
  classVariableNames: ''
  package: 'Spec-BuildUIWithSpec'
```

```
RadioButtonExample class >> defaultSpec
  ^SpecColumnLayout composed
    newRow: [:r |
      r newColumn: [:c | c add: #f1 ; add: #f2 ; add: #f3 ];
      newColumn: [:c | c add: #t1 ; add: #t2 ; add: #t3 ]];
    newRow: [:r | r add: #rinse ];
    yourself
```

```
RadioButtonExample >> extent
  ^160@150
```

In the `initializeWidgets` method, we will add the fabric radio buttons `f1` through `f3` to a `fabric` button group and the temperature buttons `t1` through `t3` to a `temperature` button group. Since these button groups are used nowhere else in the UI, we only have them as local variables of the method and not as instance variables of the class.

The code below shows the method. We first create both button groups, then the `rinse` button, and then the different temperature and fabric buttons are created and added to their group. The `rinse` button is configured to be de-selectable, and the `f1` and `t1` buttons are the default buttons of their respective groups, meaning they will be selected when the UI opens. Note that it would be better to use more meaningful variable names for clarity reasons.

```
RadioButtonExample >> initializeWidgets
  | fabric temperature |
  fabric := RadioButtonGroup new.
  temperature := RadioButtonGroup new.
```

```
  rinse := self newRadioButton.
  rinse label: 'Rinse Extra';
    canDeselectByClick: true.

  f1 := self newRadioButton.
  f1 label: 'Cotton'.
  fabric addRadioButton: f1; default: f1.

  f2 := self newRadioButton.
  f2 label: 'Synthetic'.
  fabric addRadioButton: f2.

  f3 := self newRadioButton.
  f3 label: 'Delicate'.
  fabric addRadioButton: f3.

  t1 := self newRadioButton.
  t1 label: '60'.
  temperature addRadioButton: t1; default: t1.

  t2 := self newRadioButton.
  t2 label: '40'.
  temperature addRadioButton: t2.

  t3 := self newRadioButton.
  t3 label: '30'.
  temperature addRadioButton: t3
```

Lastly, we include some logic that will trigger on button click of the different fabric buttons. If the 'Synthetic' button is selected, and the temperature is set to 60 degrees, the temperature will be lowered to 40. If the 'Delicate' button is selected, the temperature will be set to 30 degrees and the other fabric buttons will be disabled, as seen in Figure 7-1. Conversely, if the 'Delicate' button is deselected (because another button was selected), the other fabric buttons are enabled again.

```
RadioButtonExample >> initializePresenter

  f2 activationAction: [ t1 state ifTrue: [ t2 state: true ] ].
  f3 activationAction: [
    t1 disable; state: false.
    t2 disable; state: false.
    t3 state: true ].
  f3 deactivationAction: [ t1 enable. t2 enable ]
```

With this functionality, we have demonstrated the main features of `RadioButtonModel` and `RadioButtonGroup`. For additional features of these classes, we refer to their implementation source code.

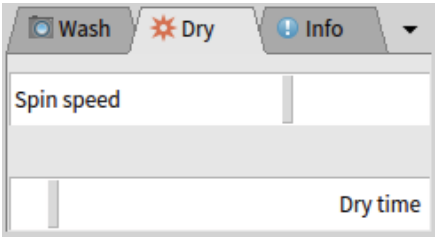

Figure 7-2 Screen shot of the washer-dryer machine control panel.

7.3 TabModel

A second set of classes that are made to work together are the `TabManagerModel` and the `TabModel` classes. In a tabbed UI, the former holds the organisation of the different tabs, and the latter represents the tabs themselves. To show how these two classes cooperate, we will extend the washing machine example from Section 7.2 to a washer-dryer machine, shown in Figure 7-2. It has a tabbed UI, were one tab is for the washing part, a second tab for the spin cycle and drying, and a third tab shows information.

To use tabs, only the `TabManagerModel` needs to be kept as an instance variable of the UI, and added to the layout:

```
ComposableModel subclass: #TabMgrExample
  instanceVariableNames: 'tabmgr'
  classVariableNames: ''
  package: 'Spec-BuildUIWithSpec'
```

```
TabMgrExample >> extent
  ^250@160
```

```
TabMgrExample class >> defaultSpec
  ^SpecLayout composed
    add: #tabmgr;
    yourself
```

To show tabs, these need to be added to the tab manager using the `addTab:` message, as shown below:

```
TabMgrExample >> initializeWidgets
  | tab |
  tabmgr := self newTabManager.
  tab := self newTab.
  tab model: RadioButtonExample new.
  tab label: 'Wash'; closeable: false;
    icon: Smalltalk ui icons smallScreenshot.
  tabmgr addTab: tab.
```

After creating the `TabManagerModel`, the code creates a `TabModel` and configures it to show an instance of the `RadioButtonExample` class. The `model:` message takes any `ComposableModel`, following the spirit of UI reuse in Spec. Lastly, additional attributes of the tab are set: a label, an icon, and the tab is made non-closable.

This is all we need to have a first version of the UI, showing the UI of Section 7.2 as a tab.

The `TabManagerModel` is dynamic: new tabs can be added and tabs can be removed when the UI is open. To do this, it is sufficient to send the messages `addTab:` and `removeTab:` to a `TabManagerModel`, with as argument the tab to be added, resp. removed. Closable tabs have a close button that allows the user to remove them from the UI as well.

To add the second part of the user interface, for drying the clothes, we extend `initializeWidgets` as follows:

```
TabMgrExample >> initializeWidgets
  [...]

  tab := self newTab.
  tab model: (self dryModel).
  tab label: 'Dry'; closeable: false;
    icon: Smalltalk ui icons smallNew.
  tabmgr addTab: tab.
```

The tab needs to be configured with an instance of a `ComposableModel`, which typically means that there is a UI class for each tab in the UI. In some cases it may however be too heavyweight to create a class for a simple UI tab. It is therefore also possible to use dynamic Spec (see Chapter 8) to create the contents of a tab.

For example, the code of `dryModel` configures a `DynamicComposableModel` to show two sliders. The first selects the maximum speed of the spin cycle, and the second how long the drying cycle should last.

```
TabMgrExample >> dryModel
  | model |

  model := DynamicComposableModel new.
  model instantiateModels: #(spin SliderModel dry SliderModel).

  model spin label: 'Spin speed'; min: 400; max: 1600; quantum: 400.
  model dry label: 'Dry time'; min: 0; max: 120; quantum: 10.

  model layout: (
    SpecColumnLayout composed
      add: #spin height: 30;  add: #dry height: 30;
      yourself).
  ^model.
```

> **Note** We strongly discourage the use of `DynamicComposableModel` for anything except the most simple UIs. Please see Section 8.3 for more considerations about the use of `DynamicComposableModel`.

Lastly, our UI contains an 'info' tab, which allows us to illustrate the ability of `TabManager` to obtain the selected tab, and take actions on tab selection. The 'info' tab will hold a `status` text field, that we keep as an instance variable and initialize before creating the tab.

```
ComposableModel subclass: #TabMgrExample
  instanceVariableNames: 'tabmgr status'
  classVariableNames: ''
  package: 'Spec-BuildUIWithSpec'
```

```
TabMgrExample >> initializeWidgets

  [...]

  self createStatus.
  tab := self newTab.
  tab model: status.
  tab label: 'Info'; closeable: false;
    icon: Smalltalk ui icons smallInfo.
  tabmgr addTab: tab.
```

In the `createStatus` method, we instantiate a `TextModel`, disable it so that the user cannot edit it, and set its initial text:

```
TabMgrExample >> createStatus

  status := TextModel new.
  status disable;
    text: 'Welcome to Washing Machine 2.0!\History: Wash ' withCRs.
```

Lastly, in `initializePresenter` we state that when a tab is selected, the label of the currently selected tab is added to the status text. This in effect creates a navigation history, completing the UI of our washer-dryer machine.

```
TabMgrExample >> initializePresenter

  tabmgr whenTabSelected: [
    status text: (String streamContents: [:s |
      s nextPutAll: status text.
      s nextPutAll: ' > '.
      s nextPutAll: tabmgr selectedTab label])].
```

7.4 Toolbars and Pop-up Menus

Both toolbars and pop-up menus in Spec are the result of a collaboration between three classes:

- A `MenuModel` containing different `MenuGroupModel` instances and displaying them, separated by a splitter.
- Multiple `MenuGroupModel` instances, each containing a number of `MenuItemModel` instances.
- Various `MenuItemModel` instances, each representing a menu item with a specific appearance and behaviour.

Spec currently does not provide straightforward support for contextual menus on the UI. It is however easy to add a toolbar to a UI, as it is just another widget that is added to the layout. For example, consider the `WatchpointWindow` class that is standard in Pharo 5. It has a menu held in its `menu` instance variable, and its layout method places the toolbar as first widget of a column of widgets:

```
WatchpointWindow >> defaultSpec
  ^ SpecColumnLayout composed
      add: #menu height: self toolbarHeight;
      add: #list;
      add: #inspectIt height: self toolbarHeight
```

The `MenuModel` class is the only class that should be instantiated by the programmer. To create a group, send `addGroup:` to a `MenuModel`, with a block as argument. This will create the group and add it to the menu. The block receives one argument: the instance of the group that was created. Similarly, sending `addItem:` to that `MenuGroupModel` creates a new `MenuItemModel` and adds it to the group. The argument of `addItem:` is also a 1-argument block, and this argument is the menu item that is created. While this explanation may sound confusing, the resulting code is quite readable, as will be shown in the example below.

`MenuItemModel` provides the following methods for configuration:

- `name:` sets the text of the menu item.
- `icon:` sets the icon.
- `description:` adds a tool-tip.
- `shortcut:` adds a shorcut for the item.
- `action:` adds a block to execute when the item is selected.
- `subMenu:` takes a `MenuModel` that represents the submenu that corresponds to this item.

Since `MenuItemModel` is meant to be used both as a toolbar and a menu item, it is possible to create menu items without text or without icons, and the tool-tip works both when used as a toolbar and as a menu item. The last two configuration options: executing an action or opening a submenu are **not** mutually exclusive. For example, in a menu hovering over an item with a

submenu opens the submenu, but still allows for the menu item itself to be selected.

To show the construction of a menu, we expand our washing machine example of the previous sections. We add a menu for technicians to use when troubleshooting or fixing the machine. To do this, we add a `menu` instance variable and accessors to the `TabMgrExample` class as well as a `populateMenu` method. The `initializeWidgets` method is appended with a `self populateMenu` line, and the code of this method is as follows:

```
TabMgrExample  >> populateMenu
  | submenu |

  menu := MenuModel new.
  submenu := MenuModel new.

  submenu addGroup: [ :group |
      group addItem: [ :item |
        item name: 'Soft Reset';
          action: [ status text: 'History: Wash '.];
          icon: Smalltalk ui icons exception ].
      group addItem: [ :item|
        item name: 'Hard Reset';
          action: [ GrowlMorph openWithContents: 'Just pull the
    plug!' ];
          icon: Smalltalk ui icons smallError ] ].
```

After defining a `submenu` temporary variable, the code above initialises both the `menu` and `submenu` variables to a fresh `MenuModel` instance. For the sake of the example, the code first constructs the submenu, adding a soft and hard reset menu item with their respective icons and actions. The code for these menu items is straightforward (we assume the `GrowlMorph` also works on our washing machine).

We now continue with the definition of the main menu. As said previously, there is no easy way to create a contextual menu that pops up when the right mouse button is clicked. It is possible to show a pop-up menu by sending `buildWithSpecAsPopup popUpInWorld` to a `MenuModel` instance, but binding this code to a right mouse click is not straightforward.

A workaround we use here is to define a menu item that pops up the menu when it is selected. This seems like creating a chicken and egg problem, since there is no way to select the menu item when the menu is not visible! This is however not the case: we can associate a shortcut key to the menu item, so pressing the shortcut reveals the menu.

The code below continues the `populateMenu` method and populates the main menu. The first item is the menu revealing item. It is triggered by pressing control-r on Windows and Linux, and command-r on Mac, as specified by

the `$r meta` key combination. This menu item also shows how to specify a tooltip. The rest of the code is straightforward.

```
menu addGroup: [ :group |
  group addItem: [ :item |
    item name: 'Reveal this menu';
      action: [menu buildWithSpecAsPopup popUpInWorld];
      description: 'This entry exits to have a shortcut for this
 menu.';
      shortcut: $r meta ].
  group addItem: [ :item |
    item name: 'Status Info';
      action: [ GrowlMorph openWithContents: tabmgr selectedTab
 label];
      icon: Smalltalk ui icons help] ].
```

Lastly, we add a second group to the main menu, causing a splitter to appear between the above two items and the below item. This last item simply shows the submenu that we created at the beginning of the method.

```
menu addGroup: [ :group |
  group addItem: [ :item |
    item name: 'Actions';
      subMenu: submenu ]].

menu applyTo: self.
```

At the very end of the method, the `applyTo:` method is sent to the menu, causing the shortcuts defined in it to be registered to the `TabMgrExample` widget. This is what makes pressing a shortcut of a menu item trigger the action of that menu item.

Nothing else is needed to associate the menu to the UI. There is no need to add the menu in a layout method, since our example does not have a toolbar widget shown. However, in case of adding a toolbar the `menu` will need to be added to the layout, as we have shown in the `WatchpointWindow` example at the beginning of this section.

Last but not least, menus are not completely static: menu items can be disabled and enabled by sending `enabled: false`, resp. `enabled: true` to them. The structure of a menu can also be changed by adding and removing items and groups. In the example above, since the menu is rebuilt on each pop-up such changes are immediately visible. In contrast, when used as a toolbar the menu is a widget, so the UI will need to be rebuilt as discussed in Chapter 8.

7.5 Conclusion

In this chapter we have shown how to configure and use more advanced widgets. We have discussed `TextModel` and its subclass `TextInputFieldModel`, `RadioButtonModel` and its grouping functionality, `TabModel` and the tab manager, and the various classes around `MenuModel`. In some examples we have used dynamic spec, and this is discussed in the next chapter.

CHAPTER 8

Dynamic Spec

Up until now we have seen user interfaces that are static: Once the UI is opened, the widgets are never changed and the layout of the UI is never changed. Also, when writing the code of the UI we know exactly which (and how many) widgets to use. There are however cases where user interfaces need to be more dynamic. For example, a file dialog can have a preview area for the currently selected file: when the file is text, its contents is shown in a text field, when it is an image, the image is shown, and so on. Another case is visualizing collections of data where we cannot determine at development time how data items will be shown. For example in a genealogy application clicking on a parent reveals information on all of the children, each child being presented using a complex, editable, widget.

Spec also provides support for these kinds of user interfaces thanks to its dynamic features, and we show them in this chapter. Firstly we present how to dynamically change a UI that is already open. Secondly we give an example of how to defer to UI opening time the choice of widgets to show, instead of hardcoding this when writing the UI. Thirdly we script a complete UI from within one (big) piece of code.

8.1 Dynamically changing an already opened UI

The first dynamic feature of Spec that we show is changing the layout and contents of an UI that is already open. This is done by using the `needRebuild:` and `buildWithSpecLayout:` methods.

- `needRebuild:` This method is used to signal that the complete UI does *not* need to be rebuilt from scratch.

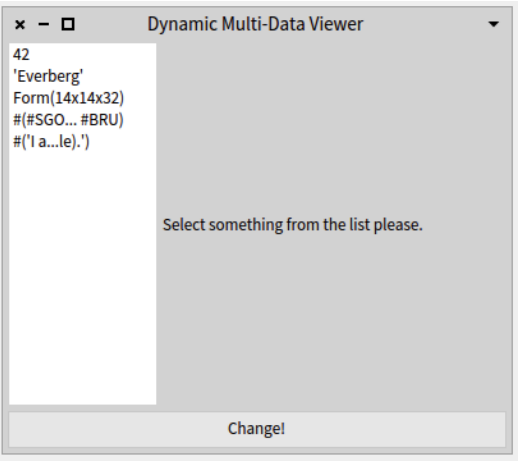

Figure 8-1 The multi-data viewer when just opened.

- `buildWithSpecLayout:` The method triggers a rebuild of the UI, which is then limited to laying out the widgets as specified in the `SpecLayout` instance given as argument.

A running example

To show the use of these two methods, we create an example user interface that is a multi-data viewer: on the left it shows a list of different kinds of data, and on the right it contains a 'view' widget that shows this data in the most appropriate manner. The multi-data viewer can also change its layout from this horizontal form to a vertical form where the list of data is on top and the view widget at the bottom. This change is done by clicking on a button. We show the multi-data viewer when just opened in Figure 8-1 and in vertical mode with an item selected in Figure 8-2.

We need to set up the UI before we can talk about changing it after it is opened. We start with the class definition, window size and title definition, and functionality for setting the contents of the list:

```
ComposableModel subclass: #DynamicViewer
  instanceVariableNames: 'list view button state'
  classVariableNames: ''
  package: 'Spec-BuildUIWithSpec'
```

```
DynamicViewer >> extent
  ^ 400@350
```

```
DynamicViewer >> title
  ^ 'Dynamic Multi-Data Viewer'
```

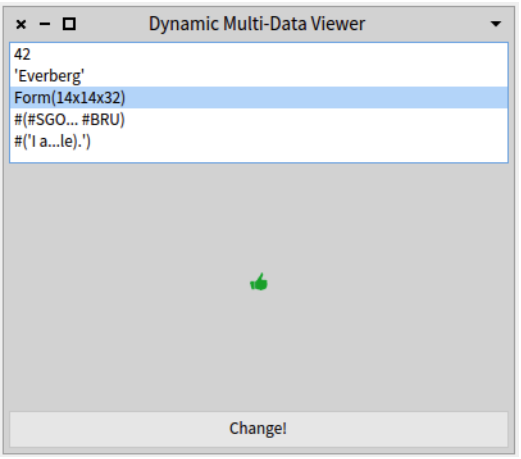

Figure 8-2 The multi-data viewer in vertical layout, with a Form selected.

```
DynamicViewer >> items: aCollection
  list items: aCollection
```

Similarly, the `initializeWidgets` method is simple. The list shows the `printString`, contracted to 15 characters. We record that the UI is using a horizontal layout by setting `state` to the `$h` character, and we set the view to the default, which is a label (as seen in Figure 8-1).

```
DynamicViewer >> initializeWidgets
  list := self newList.
  list displayBlock: [ :item | item printString contractTo: 15 ].
  button := self newButton.
  button label: 'Change!'.
  state := $h.
  view := self defaultView.
```

```
DynamicViewer >> defaultView
  | cm |
  cm := self newLabel.
  cm label: 'Select something from the list please.'.
  ^cm
```

The layout of the window is given by the `horizontalSpec` method below. Again, there are no surprises here. This is the default layout thanks to the `<spec: #default>` pragma, see Chapter 5 for more information.

```
DynamicViewer class >> horizontalSpec
  <spec: #default>
  ^ SpecColumnLayout composed
    newRow: [ :r |
      r newColumn: [:c | c add: #list] left: 0 right: 0.7.
```

```
      r newColumn: [:c | c add: #view] left: 0.32 right: 0];
    newRow: [ :r | r add: #button ] height: self toolbarHeight;
      yourself.
```

We can now already open this UI and give it a list of items to show. For example, the code below produces the UI as shown in Figure 8-1.

```
| viewer |
viewer := DynamicViewer new.
viewer openWithSpec.
viewer items: {
  42 .
  'Everberg' .
  #thumbsUp asIcon .
  #(SGO CDG ZYR BRU) .
  (OrderedCollection withAllSubclasses
    collect:[: cls | cls comment]) asArray}.
```

Changing the UI layout

Clicking on the 'Change!' button however does not do anything yet. For this we need to define the action block of the button, which we do in `initializePresenter`, as below. In this action block we first switch the state between `$h` and `$v` as appropriate. Second, we signal that the complete UI does *not* need to be rebuilt from scratch, with the `self needRebuild: false.` line. Thanks to this, the next rebuild of the UI will be limited to only performing layout operations. Third and last, the layout change is then triggered by the `buildWithSpecLayout:` message. It takes a `SpecLayout` as argument, and it will change the layout of the UI to the specified layout.

```
DynamicViewer >> initializePresenter
  button action: [
    state := (state = $v) ifTrue: [ $h ] ifFalse: [ $v ].
    self needRebuild: false.
    self buildWithSpecLayout: self currentSpec.
    ].
```

The new layout for the UI is returned by the `currentSpec` method shown below. It simply delegates to the `horizontalSpec` or `verticalSpec` methods, depending on the value saved in `state`.

```
DynamicViewer >> currentSpec.
  ^ state = $v
    ifTrue: [ self class verticalSpec ]
    ifFalse: [ self class horizontalSpec ]
```

```
DynamicViewer class >> verticalSpec
  ^ SpecColumnLayout composed
      newRow: [:r | r add: #list] top: 0 bottom: 0.7;
      newRow: [:r | r add: #view] top: 0.32 bottom: 0.02;
```

```
      newRow: [:r | r add: #button] height: self toolbarHeight;
      yourself.
```

The above is all that we need to do to be able to switch the layout of the viewer from horizontal to vertical mode. At its core lie the two calls: `self needRebuild: false` and `self buildWithSpecLayout: <XXX>`. All the rest is really just machinery used to define the value of `<XXX>`.

Replacing a widget with another one

Fundamentally, replacing a widget with another one is just a special case of changing the layout of the UI. The layout used is *actually the same*, it just so happens that one widget instance was first replaced for another.

To show this, we implement the code that visualizes the selected item of the list by showing it in the `view` widget. This is done by adding new code to `initializePresenter`, as below. The list selection change block performs the same layout operations as the button action block discussed above. The difference lies in the assignment of a new widget to the `view` instance variable. This then allows the UI to, e.g. show the `Form` as in Figure 8-1.

```
DynamicViewer >> initializePresenter
  button action: [
    state := (state = $v) ifTrue: [ $h ] ifFalse: [ $v ].
    self needRebuild: false.
    self buildWithSpecLayout: self currentSpec.
    ].

  list whenSelectedItemChanged: [ :new |
      view := self widgetFor: new.
      self needRebuild: false.
      self buildWithSpecLayout: self currentSpec.
    ].
```

The responsibility of the `widgetFor:` method is to return an instance of the correct widget, showing the selected item. For the sake of keeping this example small, we provide below some decidedly non-OO code that fundamentally implements a big switch statement over the type of the selected item. A clean OO implementation would use double-dispatch, and is left as an exercise to you, dear reader.

```
DynamicViewer >> widgetFor: aDatum
  | cm |
  aDatum isNil ifTrue: [ ^ self defaultView ].
  aDatum isForm ifTrue: [
    cm := self newImage.
    cm image: aDatum.
    ^ cm ].
  aDatum isArray ifTrue: [
    cm := self newList.
```

```
    cm items: aDatum.
    ^ cm ].

  "default case"
  cm := self newText.
  cm text: aDatum asString.
  ^cm
```

> **Note** The code for `widgetFor:` does not follow good OO practices. In the context of this chapter we choose to write this ugly code to make the example more self-contained. Any student of ours caught writing code like this will be invited to our respective offices for a long conversation.

8.2 Dynamically populating a UI with widgets

It is not always possible to determine beforehand how many widgets will be shown in a UI, and instead this needs to be decided when the UI is opened. Spec also provides support for this. Dynamic population is done by subclassing `DynamicComposableModel` (instead of `ComposableModel`), using its `assign:to:` method to dynamically create virtual instance variables for the different widgets and by defining an instance-side layout.

Adapting the multi-data viewer

To show this, we build an extension to the dynamic multi-data viewer of the previous section. It shows the contents of an array differently than simply in a list view, seen for example in Figure 8-3. We create a new `DynamicArrayViewer` widget for this detailed view, and hence first change the implementation of `DynamicViewer >> widgetFor:` to return such a widget:

```
DynamicViewer >> widgetFor: aDatum
  | cm |
  aDatum isNil ifTrue: [ ^ self defaultView ].
  aDatum isForm ifTrue: [
    cm := self newImage.
    cm image: aDatum.
    ^ cm ].
  aDatum isArray ifTrue: [
    cm := DynamicArrayViewer on: aDatum.
    cm owner: self.  " needed because we do not use 'instantiate:' "
    ^ cm].

  "default case"
  cm := self newText.
  cm text: aDatum asString.
  ^ cm
```

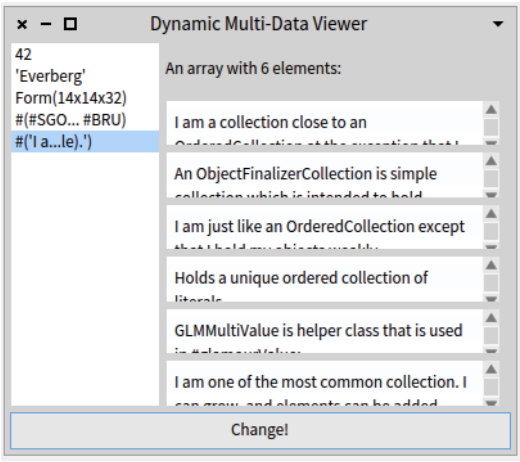

Figure 8-3 The extended multi-data viewer showing the contents of an array.

Implementing the widget for showing array contents

We now implement the widget that will show the detailed contents of an array. It is a subclass of `DynamicComposableModel`, with an instance variable `collection` that holds the array and `label` that holds the label at the top.

```
DynamicComposableModel subclass: #DynamicArrayViewer
  instanceVariableNames: 'collection label'
  classVariableNames: ''
  package: 'Spec-BuildUIWithSpec'
```

The `collection` instance variable is initialized when the widget is instantiated using the `on:` message, as shown below. This variable must be initialized before the `initialize` method is called because in `initializeWidgets` we need it to determine the items that will be shown.

```
DynamicArrayViewer class >> on: aCollection
  | inst |
  inst := self basicNew.
  inst collection: aCollection.
  inst initialize.
  ^inst.
```

The `initializeWidgets` method iterates over the collection and instantiates a `TextModel` for each of its items. This is then added to the UI using the `assign:to:` method, as shown below.

`assign:to:` takes an instance of a `ComposableModel` and a symbol that will be used to refer to that instance. Spec virtually creates a variable together with its (virtual) accessor methods. For example, the code below will cause the generation of accessors `txt_1`, `txt_2`, etc. as well as `txt_1:`, `txt_2:`, etc.

```
DynamicArrayViewer >> initializeWidgets

  label := self newLabel.
  label label: 'An array with ', collection size asString , '
    elements:'.

  1 to: collection size do: [ :count | | model |
    model := self newText.
    model text: (collection at: count) asString.
    self assign: model to: ('txt_',count asString) asSymbol ]
```

> **Note** The generation of these virtual accessors allows the widgets to be referred to in the rest of the code through an accessor, as if it was a normal variable.

Now the only thing left to do is to define a layout for the widgets of this UI. In addition to the class-side layout features of `ComposableModel`, `DynamicComposableModel` also allows a layout to be specified at instance side. This can be done either by overriding the `layout` method to return the required `SpecLayout` instance, or using the `layout:` accessor to set the layout. These two options are similar to what is possible with window title and size, as discussed in Chapter 6.

In our example, we need to define a layout at instance side since the number of widgets differs per instance. We choose to override the `layout` method, as shown below. This code creates a column and adds all widgets that were created in `initializeWidgets`. We can see that the dynamically created widgets are referred to by their name `txt_1`, `txt_2`, etc. , reconstructed by iterating over the collection. This is a sub-optimal implementation and we will show an alternative implementation at the end of this chapter.

```
DynamicArrayViewer >> layout
  | col |
  col := SpecColumnLayout composed.
  col add: #label.
  1 to: collection size do: [:count|
    col add: ('txt_',count asString) asSymbol ].
  ^ col
```

This is all that is required to dynamically populate a UI. Note that this UI can also be opened stand-alone as it is a subclass of `ComposableModel`. For example, the snippet of code below will open it with the same contents as in Figure 8-3.

```
(DynamicArrayViewer on:
       (OrderedCollection withAllSubclasses
           collect:[: cls | cls comment]))
   openWithSpec.
```

An alternative implementation of initializeWidgets

Above we have seen that `initializeWidgets` and `layout` do double work when creating the names of the virtual variables of the widgets. We can avoid this if instead of overriding the `layout` method, we use the `layout:` accessor inside of `initializeWidgets` as follows:

```
DynamicArrayViewer >> initializeWidgets
  | col |
  col := SpecColumnLayout composed.

  label := self newLabel.
  label label: 'An array with ', collection size asString , '
    elements:'.
  col add: #label.

  1 to: collection size do: [ :count | | model nam |
    model := self newText.
    model text: (collection at: count) asString.
    nam := ('txt_',count asString) asSymbol.
    self assign: model to: nam.
    col add: nam ]

  self layout: col.
```

This implementation builds the layout at the same time as the widgets are built. It adds a reference in the layout to each widget just after it is created. At the end of the method, the layout for the UI is set using the `layout:` method, removing the need to override `layout` as in the previous implementation.

8.3 Hacking together a UI in the Playground

Thanks to `DynamicComposableModel`, Spec also allows for a more scripting-like development style, where the entire UI is defined in one giant piece of code, typically built incrementally. This may be useful, e.g. for 'quick and dirty' prototyping where you want to omit the creation of a class for the UI as well as the definition of multiple methods. In this section we show how we can hack together a simple UI, scripting style, from within the Playground.

The UI we build here is a simple example showing a number and two buttons, one to increment the number and one to decrement it. This example is adapted from code present in the Pharo 5 image: `DynamicSpecExample >> openOnInteger` by Torsten Bergmann, for which we give our thanks.

We start our example code by instantiating a `DynamicComposableModel` and setting the title and size of the window for our UI by using the `title:` and `extent:` methods (see Chapter 6). Note that the local variable `num` will hold the number shown in the UI.

```
| ui num |

num := 0.
ui := DynamicComposableModel new.
ui title: 'I am dynamic'.
ui extent: 250@70.
```

We then add the widgets to the UI, using the `instantiateModels:` method. This method takes as argument a collection of pairs. The first element of the pair is the name of the widget, i.e., the name of the instance variable if this would be a normal, static, UI. The second element of the pair is a subclass of `ComposableModel`, determining the type of the widget.

```
ui instantiateModels: #(
  text LabelModel
  plus ButtonModel
  minus ButtonModel ).
```

Hence this UI will hold a `text` widget, which is a `LabelModel` instance, as well as two `ButtonModel` widgets, in `plus` and `minus`, respectively. Fundamentally, the code above is the only change with respect to what we have seen before. This is because, as in the `assign:to:` method shown in the previous section, Spec also automatically generates the (virtual) accessors for the widgets, making it possible to just use accessors in the remainder of the code.

With the widgets defined and instantiated, we can now configure them, as we would do in a normal `initializeWidgets` or `initializePresenter` method:

```
ui text label: num asString.

ui minus
  label: '-';
  state: false;
  action:   [
    num := num -1.
    ui text label: num asString ].

ui plus
  label: '+';
  state: false;
  action:   [
    num := num +1.
    ui text label: num asString ].
```

The last thing we need to do is to give the UI a layout. This is done by calling the `layout:` method, giving it an instance of a `SpecLayout` that specifies the layout. The UI can then be opened and is fully functional.

```
ui layout: (SpecLayout composed
  newColumn: [ :c |
    c
      add: #text height: 25;
      newRow: [ :r | r add: #minus ; addSplitter; add: #plus ]
  height: 25 ];
  yourself).

ui openWithSpec.
```

Note When doing this you are ignoring all the code writing and management facilities that Pharo has to offer (e.g., the browser) and not cleanly separating the different responsibilities of the code of the UI. You also hamper future extension via subclassing. In other words, this is truly a quick and dirty way to hack together a user interface that should really not make it to production.

8.4 Conclusion

In this chapter we have discussed the features of Spec that allow for more dynamic user interfaces: changing the layout and the content on-the fly, as well as determining the UIs' widgets at opening time. We ended with a last dynamic feature: allowing the complete construction of a UI in one big block of code, scripting style.

CHAPTER 9

Tips and Tricks

This chapter collects many small examples of features that may be of use when building user interfaces with Spec. For the sake of brevity, some examples use dynamic Spec (see chapter 8), but these of course also work without using the dynamic features of Spec. Most of the examples are short snippets of code that should be auto-explicative.

9.1 Integrating the different UI frameworks

In this section we show different ways in which the multiple UI frameworks present in Pharo can play together with Spec.

Include a Morph in a Spec UI

Any morph can be included as a widget in a Spec UI. To do this, send the Morph instance the `asSpecAdapter` message, it will return a Spec widget that can be used in your UI.

As an example, below we show the code that turns the `CalendarMorph` into a plain Spec UI, opened with `SpecCalendar new openWithSpec`:

```
ComposableModel subclass: #SpecCalendar
  instanceVariableNames: 'morph'
  classVariableNames: ''
  package: 'Spec-BuildUIWithSpec'
```

```
SpecCalendar >> initializeWidgets
  morph := (CalendarMorph openOn: Date today) asSpecAdapter
```

```
SpecCalendar class >> defaultSpec
  ^SpecLayout composed add: #morph; yourself
```

```
SpecCalendar >> extent
  ^220@200
```

```
SpecCalendar >> title
  ^'SpecCalendar'.
```

Include a Glamour UI in a Spec UI

User interfaces written in Glamour can also be embedded in a Spec UI thanks to a bridge class `GlamourPresentationModel`. This is a `ComposableModel` subclass that wraps Glamour user interfaces such as the Playground or the Inspector.

For this functionality to work in Pharo 5 you should first load the integration package:

```
Gofer it
    smalltalkhubUser: 'jfabry' project: 'Playground';
    package: 'Spec-Glamour';
    load
```

The `GlamourPresentationModel` by default wraps the Playground, which means that we can open a Spec window on a Glamour Playground as follows: `GlamourPresentationModel new openWithSpec`.

To use another Glamour UI, the `GlamourPresentationModel` must be configured before it is opened, sending it the `presentationClass:startOn:` message, with as arguments the class of the Glamour UI and the data to be shown in the UI. For example, below we open the inspector on 42 (in a Spec window).

```
| cm |
 cm := GlamourPresentationModel new.
 cm presentationClass: GTInspector startOn: 42.
 cm openWithSpec
```

The GT Playground is a bit unusual in the data that it opens on, this data must be a `GTPlayPage` instance, for example as below.

```
| cm |
 cm := GlamourPresentationModel new.
 cm presentationClass: GTPlayground startOn: (GTPlayPage new
    saveContent: '42').
 cm openWithSpec
```

Make a Spec presentation for the GT Inspector

The inverse of the previous tip can also be realized, in a way: It is possible to write code for a GT Inspector tab such that the tab shows a Spec UI. To do this the `display:` block of the tab should evaluate to a `ComposableModel`

instance that has been set up completely by sending it the `buildWithSpec` message.

For example, the code next is for a tab for `OrderedCollection`, showing all items in the collection as a ListModel.

```
OrderedCollection >> gtInspectorItemsAsListIn: composite
  <gtInspectorPresentationOrder: 10>
   composite spec
     title: 'AsList';
     display: [ :elt | | cm |
       cm := ListModel new.
       cm items: self.
       cm buildWithSpec.
       cm]
```

Note By combining the above with the previous tip, it is now possible to achieve Infinite Inspector Recursion between Spec and Glamour UIs:

```
Object >> gtInspectorRecursive: composite
  <gtInspectorPresentationOrder: 10>
   composite spec
     title: 'Recursion!';
     display: [ :elt | | cm |
       cm := GlamourPresentationModel new.
       cm presentationClass: GTInspector startOn: self.
       cm buildWithSpec.
       cm]
```

9.2 Lists, trees and tables

In this section we group all tips that treat lists, trees and tables.

A scrollable list of widgets

`ListModel` can show more than just text, it can also visualize any kind of widget. To do this, in the `displayBlock:` send the widgets the `buildWithSpec` message. This allows to create, e.g. a scrollable list of buttons that show the comments of the classes in the *Files* package, as shown in Figure 9-1.

```
ListModel new
  displayBlock: [ :x | x buildWithSpec ];
  items:
    ('Files' asPackage classes
      collect: [ :cls |
        ButtonModel new icon: cls systemIcon; label: cls name;
          action: [TextModel new text: cls comment; openWithSpec]]);
  openWithSpec
```

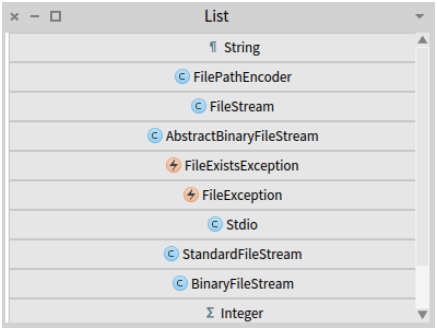

Figure 9-1 The Scrollable List of Widgets

List items do not update automatically

Adding items to a collection held in a list does *not* update the widget. A refresh only happens when the items of the list are set using the `items:` message. Note that setting the items also sorts them if a sort block is specified.

```
|container count|
 count :=0.
 container := DynamicComposableModel new.
 container instantiateModels:  #(#list ListModel #plus ButtonModel).
 container plus label: '+1'.
 container plus action: [ | items |
   count := count + 1.
   items := container list getList asOrderedCollection.
   items add: count asString.
   container list items: items ].
 container layout:
   (SpecRowLayout composed add: #list ; add: #plus ; yourself).
 container openWithSpec
```

Setting the selection in a tree

Programmatically setting the selection in a `TreeModel` does not automatically highlight the selected item, and inversely highlighting an item does not automatically select it. Therefore, you need to perform both operations to select and highlight a tree item, as in the example below. Note the last two statements in the `whenBuiltDo:` block: the first sets the selected item, and the second highlights it.

```
| tree chblock |
 tree := TreeModel new.
 chblock := [:cl | | subs node |
   subs := cl subclasses.
   node := TreeNodeModel new.
```

```
    node content: cl;
      hasChildren: [ subs isNotEmpty  ];
      children: [ subs collect: [:sub | chblock value: sub ] ] ].
  tree roots:
        (Collection subclasses collect: [:cl| chblock value: cl ]).
  tree whenBuiltDo: [ |sel|
    sel := tree roots third.
    tree selectedItem: sel.
    sel selected: true.].
  tree openWithSpec
```

Show a table-like view

There are multiple ways in which a table-like widget can be shown to the user. The most straightforward way is the `MultiColumnListModel`, for example as below. Note that the display block needs to be set to return the item itself, since the default behavior of the display block turns the array into its `printString`.

```
MultiColumnListModel new
  items: {
    {'Origin' . 'Destination' . 'Start Time' . 'Stop Time'} .
    {'Santiago' . 'Paris' . '15:30' . '11:15'} .
    {'Paris' . 'Santiago' . '23:30' . '8:00'} . };
  displayBlock: [ :x | x ];
  openWithSpec.
```

Alternatively, a `TreeModel` can be used, which allows for more fancy formatting options. (This example thanks to Nicolai Hess)

```
|r m col1 col2 col3|
 r := FileLocator vmDirectory.
 m := TreeModel new.
 m roots: r allFiles.
 m rootNodeHolder: [ :item |
    TreeNodeModel new
        content: item;
        icon: Smalltalk ui icons smallLeftFlushIcon ].
 m title: r fullName.
 col1 := TreeColumnModel new
    displayBlock: [ :node | node content basename ];
    headerLabel:'Name'.
 col2 := TreeColumnModel new
    displayBlock: [ :node | node content creationTime ];
    headerLabel:'Time'.
 col3 := TreeColumnModel new
    displayBlock: [ :node | node content permissions];
    headerLabel:'Permissions'.
  m columns: {col1. col2 . col3}.
  m openWithSpec.
```

9.3 Using the underlying widget library

In this section we show how to use some features of the underlying widget library (Morphic in Pharo 6 and before) in Spec.

> **Note** Warning: By following these tricks you are tightly coupling your UI to the underlying widget library. As a result, your UI may break if there are changes in this library, or if Spec uses another library for its underlying widgets.

Customizing the appearance of a widget

If Spec does not allow you to customize the appearance of a widget, but the underlying widget library does, you can send messages to the widget just after it has been built. For this, configure the widget with the `whenBuiltDo:` message. The argument is a block that takes one argument, which will be an instance of a `WidgetBuilt` announcement.

For example, the code below assumes the use of Morphic and results in the label being rendered in red, and using the Balloon font in Italic.

```
|container|
  container := DynamicComposableModel new.
  container instantiateModels:  #(#red LabelModel).
  container red label: 'I am red'.
  container red whenBuiltDo: [:ann|
    ann widget color: (Color red);
         font: BalloonMorph balloonFont emphasis: 2].
  container layout: (SpecLayout composed add: #red; yourself).
  container openWithSpec
```

Get called at periodic intervals

Morphic implements a 'stepping' mechanism that allows a morph to register itself for callbacks that are executed at regular intervals. This allows, e.g. for automatic refreshing of a windows' content every *n* milliseconds. Spec also provides support for this, by overriding `ComposableModel`'s method `defaultWindowModelClass` to return a `TickingWindowModel`. An example of this can be found in the Pharo 5 image in the `WatchPointWindow` class:

```
WatchpointWindow >> defaultWindowModelClass
  ^ TickingWindowModel
```

With this override set, every `stepTime` milliseconds the `step` method will be called by the Morphic infrastructure. The default implementation of `step` does nothing, and `stepTime` returns 1000. So at least `step` should be overriden to implement the required action. For example, the `WatchPointWindow` does not override `stepTime` and implements `step` as below, refreshing the list that is shown.

```
WatchpointWindow >> step
  self refreshItems
```

```
WatchpointWindow >> refreshItems
  | max values |
  values := self watchpoint values.
  max := values size.
  list items: (values copyFrom: (1 max: max - numItems) to: max)
    reversed.
```

9.4 Testing the functionality of a Spec UI

It is possible to test Spec UIs by programmatically simulating button clicks, list selections et cetera. Furthermore, this can be performed without having to open the UI at all.

For example, consider the code below that tests the three buttons of the `CustomerSatisfaction` UI of Chapter 2.

```
| cs |
 cs := CustomerSatisfaction new.
 cs buildWithSpec.
 cs buttonHappy performAction.
 self assert: [cs screen label = 'Happy' ].
 cs buttonNeutral performAction.
 self assert:[ cs screen label = 'Neutral' ].
 cs buttonBad performAction.
 self assert: [cs screen label = 'Badz' ].
```

In the third line of code, we build the UI without opening a window. Then, for each button we programmatically click it and check if the label shown on screen matches the expected value. The last test will fail because the label shown on screen sais 'Bad' and not 'Badz'.

There is however no uniform API for testing the different widgets of Spec, for each widget you must establish which method to call to programmatically simulate the widget manipulation you wish to test. For example, the code below uses list selection to test some behavior of the text field of the `ProtocolBrowser` of Chapter 3.

```
| pb |
 pb := ProtocolBrowser new.
 pb buildWithSpec.
 pb viewer models list setSelectedIndex: 1.
 self assert: [ pb text text isEmpty ].
 pb viewer api methods setSelectedIndex: 1.
 self assert: [(pb text text copyFrom: 1 to: 6) = 'action' ].
```

Note The previous code is not clean because it heavily relies on the internal structure of the `ProtocolBrowser` and of the widgets that it reuses. Instead, it would be better to create an API for testing that exposes the internals that are required to be able to write UI tests. A discussion of such API creation is however out of the scope of this text.

Note If it is required have the UI open while doing the tests, change the send of `buildWithSpec` to `openWithSpec` and add a `pb window close` at the end of the code, this will open, respectively close the UI.